Fast Algorithms for the Optimum-Path Forest-based Classifier

Presentado por:

Aldo Paolo Culquicondor Ruiz

Para optar por el Título Profesional de:

Licenciado en Ciencia de la Computación

Orientador: Ph.D. José Eduardo Ochoa Luna
Co-orientador: Ph.D. (c) César Christian Castelo Fernández

Arequipa, abril de 2018

To my parents for always supporting me, even from distance, and to each of the professors, classmates and friends that I met along my career. Each of them inspired me and motivated me in a particular way to continue on the construction of my future.

Acknowledgements

To Prof. César Castelo-Fernandez, who was my advisor during the first half of my research and also helped me all the way along. I thank him also for encouraging me to write and publish my very first paper.

To Prof. João Paulo Papa, the author of the original OPF algorithms, who was an insightful reviewer of my work.

To Prof. Alexandro Baldassin and his student João Paulo Labegalini, who helped me run the experiments on very large datasets and improve the performance analysis of the parallel algorithms.

To Prof. Manuel Loaiza, who told me what to look for and helped me identify the questions that needed to be answered to get my work to its fullest potential.

Finally, to my parents, colleges and professors, who always believed in me. Thanks for all your encouragement.

Abstract

Pattern Recognition applications deal with ever increasing datasets, both in size and complexity. In this work, we propose and analyze efficient algorithms for the Optimum-Path Forest (OPF) supervised classifier. This classifier has proven to provide results comparable to most well-know pattern recognition techniques, but with a much faster training phase. However, there is still room for improvement. The contribution of this work is the introduction of spatial indexing and parallel algorithms on the training and classification phases of the OPF supervised classifier.

First, we propose a simple parallelization approach for the training phase. Following the traditional sequential training for the OPF, it maintains a priority queue to compute best samples at each iteration. Later on, we replace this priority queue by an array and a linear search, in the aim of using a more parallel-friendly data structure. We show that this approach leads to more temporal and spatial locality than the former, providing better speedups. Additionally, we show how the use of vectorization on distance calculations affects the overall speedup and we provide directions on when to use it.

For the classification phase, we first aim to reduce the number of distance calculations against the classifier samples and, then, we also introduce parallelization. For this purpose, we elaborate a novel theory to index the OPF classifier in a metric space. Then, we use it to build an efficient data structure that allows us to reduce the number of comparison with classifier samples. Finally, we propose its parallelization, leading to a very fast classification for new samples.

Resumen

Las aplicaciones de Reconocimiento de Patrones manejan conjuntos de datos en constante crecimiento, tanto en tamaño como en complejidad. En este trabajo, proponemos y analizamos algoritmos para el clasificador supervisado basado en Bosque de Caminos Óptimos (OPF, por sus siglas en inglés). Este clasificador ha probado ser capaz de proveer resultados comparables a técnicas de reconocimiento de patrones más conocidas, pero con una fase de entrenamiento mucho más rápida. Sin embargo, aun hay lugar para mejoras. La contribución de este trabajo es la introducción de indexación espacial y algoritmos paralelos en las fases de entrenamiento y clasificación del clasificador supervisado OPF.

En primer lugar, proponemos un abordaje simple de paralelización para la fase de entrenamiento. Siguiendo el entrenamiento secuencial tradicional del OPF, mantiene una cola de prioridad para calcular las mejores muestras en cada iteración. Posteriormente, reemplazamos la cola de prioridad por un arreglo y una búsqueda lineal, con el objetivo de usar una estructura de datos más adecuada para el paralelismo. Mostramos que este abordaje lleva a una mayor localidad temporal y espacial que el anterior, proveyendo mejores tiempos de ejecución. Adicionalmente, mostramos cómo el uso de la vectorización en el cálculo de distancias afecta el tiempo de ejecución y proveemos guías para su uso adecuado.

Para la fase de clasificación, primero buscamos reducir el número de cálculos de distancia respecto a las muestras del clasificador y luego introducimos un esquema de paralelización. Con este objetivo, desarrollamos una nueva teoría para indexar el clasificador OPF en un espacio métrico. Luego, la usamos para construir una estructura de datos eficiente que nos permite reducir el número de cálculos de distancia con muestras del clasificador. Finalmente, proponemos su paralelización, obteniendo una clasificación muy rápida para muestras nuevas.

Contents

1 Introduction **2**

1.1 Motivation and Context 2

1.2 Problem Statement 3

1.3 General Objective 3

1.4 Specific Objectives 3

1.5 Organization 4

2 Theoretical Foundations and Related Work **5**

2.1 The Optimum Path Forest Classifier 5

2.1.1 Training 6

2.1.2 Testing 8

2.1.3 Learning 8

2.2 Related Work 9

2.2.1 The Training Algorithm 9

2.2.2 The Classification Algorithm 10

3 Parallel Algorithms for the OPF Training **11**

3.1 A First Parallelization 11

3.2 Removing the Priority Queue 12

3.3 Vectorization of distance calculations 14

4 Spatial Indexing for the OPF Classification **16**

4.1 Spatial Indexing of the OPF 16
4.1.1 OPF-distance 17
4.1.2 OPF Triangle Inequality 17
4.1.3 The OPF Tree 18

5 Results **22**
5.1 Datasets 22
5.2 The Test Environments 23
5.3 Methodology and Results for the Training Algorithm 23
5.3.1 Execution time 24
5.3.2 Speedup 27
5.4 Methodology and Results for the Classification Algorithm 29
5.4.1 Number of Distance Calculations 29
5.4.2 Execution Time 33
5.4.3 OPF Tree-friendly 33
5.4.4 OPF Tree-unfriendly 33
5.4.5 OPF Tree-oblivious 35

6 Conclusions **37**

Appendices **39**

A Proofs for the OPF Spatial Indexing **40**
A.1 OPF Triangle Inequality 40
A.2 The OPF Tree 42

Bibliography **43**

Acronyms

OPF Optimum Path Forest

IFT Image Foresting Transform

MST Minimum Spanning Tree

CUDA Compute Unified Device Architecture

ANN Artificial Neural Network

SVM Support Vector Machine

OpenMP Open Multi Processing

GPU Graphics Processing Unit

AVX Advanced Vector Extensions

VPT Vantage-Point Tree

List of Tables

5.1 Description of the datasets used in the experiments 22

5.2 Machine/System configuration used in the experiments for the training algorithms 23

5.3 Machine/System configuration used in the experiments for the classification algorithms 24

List of Figures

2.1 An example of the OPF training algorithm 6

4.1 Visual representation of the OPF Tree construction and query 19

5.1 Execution time for the training algorithms on the SIMD-friendly datasets . 25

5.2 Execution time for the training algorithms on the SIMD-unfriendly datasets 26

5.3 Execution time for the training algorithms on the SIMD-oblivious datasets 27

5.4 Speedup results for the training algorithms on the Xeon™ (a) and Opteron™ (b) machines 28

5.5 Number of Distance Calculations for the classification algorithms on Low Intrinsic Dimensionality Datasets 30

5.6 Number of Distance Calculations for the classification algorithms on Medium Intrinsic Dimensionality Datasets 30

5.7 Number of Distance Calculations for the classification algorithms on High Intrinsic Dimensionality Datasets 31

5.8 Intrinsic dimensionality of datasets vs. the number of distance calculations on classification algorithms 32

5.9 Execution times for the classification algorithms on the OPF Tree-friendly datasets 34

5.10 Execution times for the classification algorithms on the OPF Tree-unfriendly datasets 35

5.11 Execution times for the classification algorithms on the OPF Tree-oblivious datasets 36

Chapter 1

Introduction

1.1 Motivation and Context

Nowadays, datasets in real applications contain millions, if not billions, of elements where each of them is a complex entity, requiring several features to be represented. Objects can be of different nature depending on the domain, such as documents, images, sounds, database records, etc. In order to simplify these enormous datasets and extract meaningful information for decision making, patterns need to be found and elements need to be classified.

Even with state of the art techniques in pattern recognition, in some cases we need to wait days or even weeks to obtain results. For some industries or particular domains, this can be acceptable. In general, however, we require results within a few minutes or seconds. Techniques widely used are primarily based on Artificial Neural Network (ANN) and Support Vector Machine (SVM).

Recently, a new framework to the design of graph-based classifiers, named Optimum Path Forest (OPF) has been introduced to the scientific community. Such framework comprises supervised, semi-supervised and unsupervised learning algorithms (J. P. Papa & Falcão, 2008; J. P. Papa, Falcão, & Suzuki, 2009; J. P. Papa, Falcão, De Albuquerque, & Tavares, 2012). OPF is parameterless and it can deal with non-separable data. It has obtained recognition results comparable, or even more accurate, than those of the SVMs and ANNs in a number of different applications, but usually with a much faster training algorithm. However, it can still be time consuming for very large datasets. Also, the classification time grows linearly with the size of the training dataset, thus being slower than those of the SVM's and ANN's, in general.

On the other hand, processing power is increasing in the industry, the academia and domestic computing. Companies and universities own computers equipped with multiple processors. Desktop computers and cellphones now come bundled with processors of up to 8 cores and graphic processing units with much higher concurrency levels. Several implementations of the traditional pattern recognition techniques have been made to take advantage of these computer resources. To the best of our knowledge, at the starting time of our research, implementations of OPF have not made use of parallel techniques.

1.2 Problem Statement

Supervised Learning based on Optimum Path Forest (OPF classifier from now on) represents the dataset as a complete graph, where each element is a node and the weight of the edge between them is equal to the distance in some feature space. The technique comprises three algorithms (J. P. Papa et al., 2009):

- Training: Given a path-cost function, an Optimum Path Forest is built from a set of training samples correctly labelled. A set of prototypes is chosen and optimum paths are built in an incremental manner using the Image Foresting Transform (IFT) algorithm.
- Learning: Initial testing is made by classifying a second set of correctly labelled samples. Misclassified samples are interchanged with samples of the initial training set and a second training is performed. This process is repeated until the accuracy stabilizes.
- Classification: It is done by adding the testing sample to the graph and extending the Optimum Path Forest to include it.

The training phase is in general much faster than those of the SVMs and ANNs, but there is still room for improvement. The incremental construction of optimum paths from the set of prototypes is independent for each node. Thus, the algorithm could be efficiently performed in parallel if a proper data structure is used.

On the other hand, the classification is usually slower than those of the other techniques. This is because the algorithm, in order to obtain the optimum path, calculates the path cost from every node in the training set to the testing sample. By taking advantage of the properties of some distance functions, an index can be built. As a result, some nodes could be discarded earlier during classification, thus speeding up the algorithm.

1.3 General Objective

Design efficient algorithms to speed up the Optimum Path Forest Classifier.

1.4 Specific Objectives

- Design and implement parallel training algorithms for the OPF classifier.
- Compare the proposed training algorithms against the original training process of the OPF classifier.
- Design and implement an index for the classification algorithm of the OPF classifier.
- Compare the proposed classification algorithms against the original classification process of the OPF classifier.

1.5 Organization

The remaining chapters of this thesis is organized in five chapters.

Chapter 2 comprises a review of the theoretical foundations and related work. It starts by covering the theory and original algorithms for the Optimum-Path Forest-based Classifier. Then, it presents approaches that aim to speed up both its training and classification algorithms.

In Chapter 3, we present our proposal of parallel algorithms targeting the training phase of the OPF classifier, along with the observations that led us to their design. Similarly, Chapter 4 presents the design of a new data structure for indexing the OPF classifier and speed up the classification phase.

Experiments and comparisons among the proposed approaches and the original OPF are presented in Chapter 5. The comparisons comprise number of operations and runtime performance.

Finally, Chapter 6 contains our conclusions for this work.

Chapter 2

Theoretical Foundations and Related Work

We start this chapter presenting the complete framework of the OPF classifier, which comprises three phases: training, learning and classification. Then, we present and discuss approaches related to this work that attempt to speed up the training and classification algorithms.

2.1 The Optimum Path Forest Classifier

The Optimum-Path Forest-based (OPF) supervised learning is a relatively new pattern recognition (Duda, Hart, & Stork, 2012) technique proposed by Papa et al. (J. P. Papa et al., 2009; J. P. Papa, Cappabianco, & Falcão, 2010; J. P. Papa et al., 2012; Iwashita, Papa, et al., 2014). The technique is fast, simple, multi-class, parameter independent and it doesn't make assumptions about the shape of classes in the feature space.

OPF is in fact a framework to the design of graph-based classifiers. This means that the user can design his/her own optimum-path forest-driven classifier by configuring three main modules: (i) *adjacency relation*, (ii) *methodology* to estimate prototypes, and (iii) *path-cost function*. Since OPF models the problem of pattern recognition as a graph partition task, it requires an adjacency relation to connect nodes (i.e. feature vectors extracted from dataset samples). Further, OPF rules a competition process among prototype samples, which are the most representative samples from each class. Therefore, a careful procedure to estimate them would be wise. Each prototype becomes a root of its own optimum-path tree and attempts to add other samples to it – a process which we call *conquest* – by offering them a reward. The reward is encoded by the path-cost function. At the end, a set of optimum trees, according to the path-cost function and rooted at each prototype, is obtained.

The original proposed implementation of OPF (J. P. Papa et al., 2009, 2012) employs a fully connected graph, and a path-cost function that computes the maximum arc-weight along a path. For the sake of clarity, we shall refer to this version as OPF only.

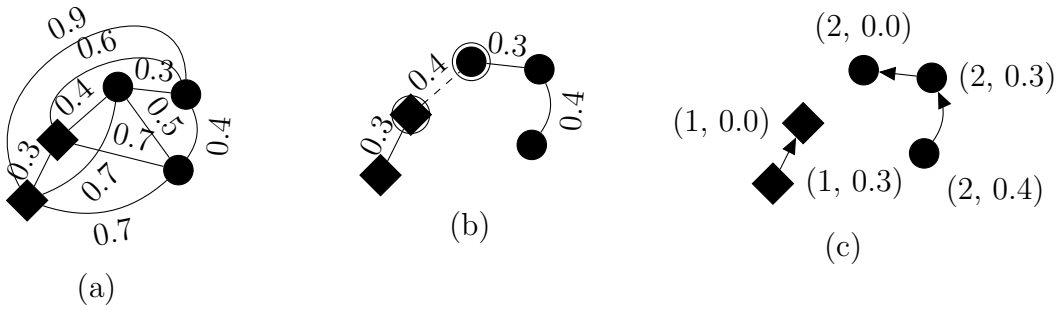


Figure 2.1: (a) Complete graph with Z_1. (b) MST and set of prototypes. (c) OPF annotated with costs and labels.

Let $\mathcal{Z}$ be a dataset whose correct labels are given by a function $\lambda(x)$, for each sample $x \in \mathcal{Z}$. Thus, $\mathcal{Z}$ can be partitioned into a training ($\mathcal{Z}_1$), validation ($\mathcal{Z}_2$) and testing ($\mathcal{Z}_3$) set. Also, we can derive a graph $\mathcal{G}_1 = (\mathcal{V}_1, \mathcal{A}_1)$ from the training set, where $\mathcal{A}_1$ stands for an adjacency relation known as *complete graph*, i.e. one that has a fully connected graph where each pair of samples in $\mathcal{Z}_1$ is connected by an edge. Additionally, each node $\mathbf{v}_i^1 \in \mathcal{V}_1$ concerns the feature vector extracted from sample $x_i^1 \in \mathcal{Z}_1$. All arcs are weighted by the distance $d : \mathcal{Z} \times \mathcal{Z} \to \mathbb{R}$ among their corresponding graph nodes. A similar definition can also be applied to the validation and test sets.

The OPF proposed by Papa et al. (J. P. Papa et al., 2009) comprises two distinct phases: (i) *training* and (ii) *testing*. The former step is based upon $\mathcal{Z}_1$, meanwhile the test phase aims at assessing the effectiveness of the classifier learned during the previous phase over the testing set $\mathcal{Z}_3$. Additionally, a *learning* algorithm was proposed to improve the quality of samples in $\mathcal{Z}_1$ by means of interchanging samples with an additional set $\mathcal{Z}_2$.

2.1.1 Training

The training step aims at building the optimum-path forest upon the graph $\mathcal{G}_1$ derived from $\mathcal{Z}_1$ (Figure 2.1a). Essentially, the forest is the result of a competition process among prototype samples that end up partitioning $\mathcal{G}_1$.

Let $\mathcal{S} \subseteq \mathcal{Z}_1$ be a set of prototypes, which can be chosen at random or using some other specific heuristic. In fact, Papa et al. (J. P. Papa et al., 2009) argued that a random choice may not be a good idea, since it can directly influence the performance of the classifier. They propose to find the set of prototypes that minimizes the classification error over $\mathcal{Z}_1$, obtaining $\mathcal{S}^* \subseteq \mathcal{Z}_1$. Such set can be found by computing a Minimum Spanning Tree (MST) $\mathcal{M}$ from $\mathcal{G}_1$, and then marking as prototypes each pair of samples (x_1, x_2), adjacent in $\mathcal{M}$, such that $\lambda(x_1) \neq \lambda(x_2)$ (Figure 2.1b). Roughly speaking, the main idea is to protect the boundary of the classes by placing prototypes there, since these regions are more prone to classification errors.

Further, the competition process takes place in $\mathcal{Z}_1$, where prototype nodes in $\mathcal{S}^*$ become the roots of their own tree and they compete to add to it the remaining samples in $\mathcal{Z}_1 \setminus \mathcal{S}^*$. We say that a node added to a prototype's tree was *conquered* by it. The

competition is based on a reward-compensation procedure, where the prototype that offers the *minimum cost* is the one that will *conquer* the sample, i.e. add it to its tree. The reward is computed based on a path-cost function, which should be *smooth*, as defined by Falcão et al. (Falcão, Stolfi, & de Alencar Lotufo, 2004). Therefore, Papa et. al. (J. P. Papa et al., 2009) proposed to use f_{max} as the path-cost function, defined as follows:

$$\begin{aligned} f_{max}(\langle \mathbf{s} \rangle) &= \begin{cases} 0 & \text{if } \mathbf{s} \in \mathcal{S}^* \\ +\infty & \text{otherwise,} \end{cases} \\ f_{max}(\pi_s \cdot (\mathbf{s}, \mathbf{t})) &= \max\{f_{max}(\pi_s), d(\mathbf{s}, \mathbf{t})\}, \end{aligned} \tag{2.1}$$

where $\pi_s \cdot (\mathbf{s}, \mathbf{t})$ stands for the concatenation between path π_s and arc $(\mathbf{s}, \mathbf{t}) \in \mathcal{A}_1$. Also, a path π_s is a sequence of adjacent and distinct nodes in $\mathcal{G}_1$ with terminus at node $\mathbf{s} \in \mathcal{Z}_1$. Notice that a *trivial path*, i.e. a single-node path, is denoted by $\langle \mathbf{s} \rangle$.

In short, by computing Equation 2.1 for every sample $\mathbf{s} \in \mathcal{Z}_1$, we obtain a collection of optimum-path trees (OPTs) rooted at $\mathcal{S}^*$, the optimum-path forest. A sample that belongs to a given OPT means it is more strongly connected to it than to any other in $\mathcal{G}_1$ (Figure 2.1c). Note that more than one path can offer the same cost to a sample. As a result, the OPF is not necessarily unique.

Input: A training set Z_1 labelled by λ
Output: The Optimum-Path-Forest P_1 as a predecessor map, a cost map C_1, a label map L_1 and the order set Z_1'

$Z_1' \leftarrow \emptyset$;
Calculate the set of prototypes $S \subset Z_1$ from MST;
foreach $s \in Z_1 \setminus S$ **do**
  assign $C_1(s) \leftarrow +\infty$;
Priority queue $Q \leftarrow \emptyset$;
foreach $s \in S$ **do**
  $C_1(s) \leftarrow 0$, $P_1(s) \leftarrow nil$, $L_1(s) \leftarrow \lambda(s)$;
  Insert s in Q;
while $Q \neq \emptyset$ **do**
  Remove from Q sample s such that $C_1(s)$ is minimum;
  Insert s in Z_1';
  foreach $t \in Z_1$ *such that* $t \neq s$ *and* $C_1(t) > C_1(s)$ **do**
    $cst \leftarrow \max\{C_1(s), d(s,t)\}$;
    if $cst < C_1(t)$ **then**
      If $C_1(t) \neq +\infty$, then remove t from Q;
      $P_1(t) \leftarrow s$, $L_1(t) \leftarrow L_1(s)$, $C_1(t) \leftarrow cst$;
      Insert t in Q;
return classifier $[P_1, C_1, L_1, Z_1']$;

Algorithm 1: Training Algorithm for OPF classifier

The training process is summarized in Algorithm 1 (J. P. Papa et al., 2009, 2012). The ordered set Z_1' was introduced in (J. P. Papa et al., 2010) to avoid comparisons against all nodes during classification.

2.1.2 Testing

In the testing step, each sample $\mathbf{t} \in \mathcal{Z}_3$ is classified individually as follows: $\mathbf{t}$ is connected to all training nodes from the optimum-path forest learned in the training phase, and the node $\mathbf{s}^* \in \mathcal{Z}_1$ that conquers $\mathbf{t}$ is obtained as the one that satisfies the following equation:

$$\mathbf{s}^* = \arg\min_{\mathbf{s} \in \mathcal{Z}_1} \max\{C(\mathbf{s}), d(\mathbf{s}, \mathbf{t})\}. \tag{2.2}$$

The classification step simply assigns the label of $\mathbf{s}^*$ as the label of $\mathbf{t}$ and its cost $C(\mathbf{t})$.

$$C(\mathbf{t}) = \max\{C(\mathbf{s}^*), d(\mathbf{s}, \mathbf{t})\} \tag{2.3}$$

As proposed in (J. P. Papa et al., 2010), if nodes are evaluated in order of Z_1', the process can finish before evaluating all nodes, as shown in Algorithm 2. The same process is used to classify sets $\mathcal{Z}_2$ and $\mathcal{Z}_3$.

Input: Classifier $[P_1, C_1, L_1, Z_1']$, evaluation set Z_2 (or testing set Z_3)
Output: Labels L_2 (or L_3) and predecessor map P_2 for Z_2 (or Z_3)

```
foreach t ∈ Z2 do
    i ← 1, mincost ← max{C1(ki), d(ki, t)};
    L2(t) ← L1(k1) and P2(t) ← ki;
    while i < |Z1'| y mincost > C1(ki+1) do
        tmp ← max{C1(ki+1, d(ki+1, t)};
        if tmp < mincost then
            mincost ← tmp;
            L2(t) ← L(ki+1) and P2(t) ← ki+1;
return [L2, Z2];
```

Algorithm 2: Classification Algorithm for OPF

2.1.3 Learning

The accuracy of the classifier can be improved by means of interchanging some samples from training set $\mathcal{Z}_1$ with samples from validation set $\mathcal{Z}_2$ (J. P. Papa et al., 2009). For this purpose, a first classifier is obtained from $\mathcal{Z}_1$ and then it used to classify samples from $\mathcal{Z}_2$, for which we know their correct classes. Samples wrongly classified from $\mathcal{Z}_2$ are interchanged with non-prototype samples from $\mathcal{Z}_1$, and the training algorithm is run again. The process is repeated until the accuracy of the classifier stabilizes. This learning procedure is based on the assumption that the most informative samples are obtained from errors.

To observe the improvement of the classifier, its precision needs to be quantified. Let c be the number of classes and $N\mathcal{Z}_2(i)$ for $i = 1, 2, \ldots, c$ the number of samples in $\mathcal{Z}_2$ with class i. We define the error $E(i)$ for class i as:

$$E(i) = \frac{FP(i)}{|\mathcal{Z}_2| - |N\mathcal{Z}_2(i)|} + \frac{FN(i)}{|N\mathcal{Z}_2(i)|}, \tag{2.4}$$

for $i = 1, 2, \ldots, c$, where $FP(i)$ and $FN(i)$ is the number of false positives and false negatives for class i, respectively. Then, the accuracy Acc of a classifier is defined by:

$$Acc = \frac{2c - \sum_{i=1}^{c} E(i)}{2c} = 1 - \frac{\sum_{i=1}^{c} E(i)}{2c}. \tag{2.5}$$

The importance of this measure is that it accounts not just for hits of the classifier, but also for misclassifications. The learning procedure aims at improving this accuracy for the OPF classifier.

2.2 Related Work

Given the proven applicability of the OPF classifier to solve Pattern Recognition problems, some attempts have been made to speed up each phase of the classifier. The approaches include the exploitation of theoretical properties, reducing redundancy and the implementation on massively-parallel hardware.

In the following sections, we present and discuss the related work existent on the training algorithm and the classification algorithm, respectively.

2.2.1 The Training Algorithm

Iwashita et. al. (Iwashita et al., 2012; Iwashita, Papa, et al., 2014) take advantage of the relationship of the MST and the OPF to speed up the training algorithm. They propose to calculate both simultaneously. Given the MST of the graph, an optimum path forest for f_{max} is obtained by removing the edges between nodes of different classes, marking them as prototypes and updating the cost and predecessor maps along the MST. Their experiments show improvements on running time of up to 77.67%. The authors do not claim to obtain a classifier (forest) with the same accuracy results, but similar. In comparison, our approach guarantees that it calculates an OPF with the same characteristics and accuracy as the original algorithm. Recall that the OPF is not unique.

During our research, a parallel algorithm for the OPF training emerged. Iwashita et. al. (Iwashita, Romero, Baldassin, Costa, & Papa, 2014) presented a fine-grained parallelization of the OPF training for Graphics Processing Unit (GPU), implemented in Compute Unified Device Architecture (CUDA).

The approach establishes a similarity between the conquering process of samples and a vector-matrix multiplication. A thread is allocated for each cell of a feature matrix to perform a single multiplication, as part of the distance calculation between samples. Then, new threads are allocated to merge the partial results. Thurst primitives (Hoberock & Bell, 2010) are used to obtain a sample with minimal cost at each step of the algorithm.

In datasets with considerable number of features, the approach obtained up to 14 times of speedup. However, the approach is slower than the serial training algorithm for datasets with a small number of features, due to the throughput of data interchange between CPU and GPU. Keeping the thread creation and synchronization to a minimum, our coarse-grained parallel algorithm in CPU obtained speedups for any number of features with different levels of parallelism.

2.2.2 The Classification Algorithm

One proposed approach for reducing the classification time involves the size reduction of the OPF. It does so by removing *irrelevant* samples, i.e. samples containing redundant information, from Z_1 (J. P. Papa et al., 2012). After performing training and learning, nodes from Z_1 which were not used for classification of Z_2 samples are marked as irrelevant and moved to Z_2. Then, training and learning is performed again. The process is repeated while precision loss is above an specified threshold. Note that, by performing more training calls, this approach increases the learning time of the classifier in exchange for a faster classification.

Romero et. al. (Romero, Iwashita, Papa, Souza, & Papa, 2014) present a fine-grained parallelization of the OPF classification targeting GPUs. They use the same vector-matrix multiplication association as the GPU fine-grained training algorithm for GPU discussed in the previous section (Iwashita, Romero, et al., 2014). As such, this approach is not suited for datasets with an small number of features.

Diniz et. al. (Diniz, Fremont, Fantoni, & Nóbrega, 2017) present a SoC/FPGA design and implementation of the OPF for embedded applications, presenting a hardware converted classification algorithm. They obtain acceleration from 2.8 to 9 times compared to an embedded processor software implementation. Algorithm 2 is implemented in an Elementary Processor (EP) to perform the classification of a set of samples in a SIMD fashion. Additionally, the distance calculation between samples is performed in a specific hardware module.

Chapter 3

Parallel Algorithms for the OPF Training

In the construction of an OPF, we always deal with a complete graph. This observation highly reduces the impact of unstructured problems that general graphs impose and gives the opportunity to implement data partitioning by vertices.

In this chapter, we present two coarse-grained shared-memory parallel algorithms for the OPF training, along with vectorized versions. One of these algorithms, the POPF, has already been published in a conference (Culquicondor, Castelo-Fernandez, & Papa, 2016). We provide implementation details using the OpenMP API. Given its wide support on different platforms and architectures, we guarantee the applicability of the algorithms to a wide range of environments.

3.1 A First Parallelization

The sequential training algorithm for the OPF classifier can be parallelized by means of three observations related to the independence of the conquest process of a node's neighbors, the completeness of the graph and the relationship between Prim's algorithm and OPF.

The first observation concerns the conquest process that takes place at each iteration of the Algorithm 1 (lines 10 to 17). In each such iteration, an unvisited sample $s \in Q$ with minimum cost $C(\mathbf{s})$ is taken, which then will attempt to conquer all its neighbors $\mathbf{t}$. It is important to notice is that the conquest of a neighbor t by s is independent of any other neighbor. Thus, the computational load for conquering neighbors could be split among threads. However, once a sample t is conquered, its cost $C(\mathbf{t})$ needs to be updated, which requires its priority in the priority queue Q to be updated as well (lines 15 to 17). LibOPF (J. Papa, Falcão, & Suzuki, 2014) uses a binary heap as the priority queue data structure, as suggested in (Falcão et al., 2004). This data structure is typically not prepared for concurrent update, thus its update process is a critical section that needs to be protected with a *mutex* in a concurrent environment. Furthermore, this data structure introduces

a $\mathcal{O}(\log(n))$ overhead in each update, where $n = |\mathcal{Z}_1|$.

The second observation concerns the graph, which is fully connected, implying that, for each node $\mathbf{s}$, its neighbors are all $t \in \mathcal{Z}_1 \setminus \{\mathbf{s}\}$. That is why the computation of the OPF along with the cost map C takes $\theta(n)^2$ operations in total for $n = |\mathcal{Z}_1|$. However, the completeness of the graph allows us to implement a simple data-partition scheme in which each thread $\delta_i, \forall i = 1, \ldots, m$ (where m is the number of threads), explores a subset $\mathcal{Z}_{(1,i)}$, such that $\mathcal{Z}_1 = \mathcal{Z}_{(1,1)} \cup \cdots \cup \mathcal{Z}_{(1,m)}$ and $\mathcal{Z}_{1,i} \cap \mathcal{Z}_{1,j} = \emptyset, \forall i \neq j$. All $\mathcal{Z}_{(1,i)}$ should be almost equally sized for the best workload distribution. Then, each thread updates the costs of each $\mathbf{t} \in \mathcal{Z}_{(1,i)} \setminus \{\mathbf{s}\}$ according to Equation 2.1 for f_{max} using arc $(\mathbf{s}, \mathbf{t})$.

The third observation is related to Prim's algorithm, which is used to calculate the Minimum Spanning Tree over $\mathcal{Z}_1$. As a matter of fact, we can use the very same OPF algorithm with a different path-cost function – i.e., the sum of weights along the path – to compute the MST (J. P. Papa et al., 2009). Therefore, the aforementioned ideas can be applied to compute the MST as well, taking advantage of parallelism in all the steps of the training process.

These observations lead us to Algorithm 3, which we call **POPF-PQ**, because it maintains the priority queue from the serial algorithm. Note that even though parallelization takes place only during the conquest process from $\mathbf{s}$, it is better to start all threads just once at the beginning of the algorithm.

The proposed approach was efficiently implemented using Open Multi Processing (OpenMP) (Dagum & Enon, 1998), a well-known API for shared-memory parallel programming. OpenMP pragmas used in the implementation are included as comments.

Note that the partitioning of the set of neighbors happens once per iteration. Launching threads is costly and, given that it would be needed $O(n)$ times, the performance of our algorithm would degrade. However, instead of launching the threads in each iteration, we can do it only once at the beginning of the algorithm. The threads are synchronized through barriers, shared variables and operations exclusive to the main thread. Then, the threads are reused for the loop in lines 15-20 in Algorithm 3.

3.2 Removing the Priority Queue

As we have proposed in (Culquicondor et al., 2016), we can improve the running time of the algorithm even further by noting that, during the pass through the neighbors of sample $\mathbf{s}$, we are indeed visiting all samples $\mathbf{t} \in \mathcal{Z}_1$. Thus, we can take advantage of this loop (line 12 in Algorithm 1) to calculate the sample $\mathbf{s}^*$ – the sample with lowest cost – to be used in the next iteration. We can also do this in separate threads by making each of them compute the node with minimum cost $\mathbf{s}_i^* \in \mathcal{Z}_{(1,i)}$. Afterwards, the main thread finds the node $\mathbf{s}^*$ with minimum cost among all $\mathbf{s}^{(*,i)}, \forall i = 1, \cdots, m$, where m is the number of threads. Such node $\mathbf{s}^*$ has the same properties as the one that the priority queue would provide if we were using it. Therefore, by using m threads, the overall time complexity of the training algorithm is quantified by $\theta(n^2/m)$.

Input: A training set Z_1 labelled by λ
Output: The Optimum-Path-Forest P_1 as a predecessor map, a cost map C_1, a label map L_1 and the order set Z'_1

$Z'_1 \leftarrow \emptyset$
Calculate the set of prototypes $S \subset Z_1$ from MST
foreach $s \in Z_1 \setminus S$ **do**
  assign $C_1(s) \leftarrow +\infty$
Priority queue $Q \leftarrow \emptyset$
foreach $s \in S$ **do**
  $C_1(s) \leftarrow 0$, $P_1(s) \leftarrow nil$, $L_1(s) \leftarrow \lambda(s)$
  Insert s in Q
in parallel for threads $i = 1, 2, \ldots, m$ `# omp pragma: parallel`
while $Q \neq \emptyset$ **do**
  in main thread Remove from Q sample s with minimum $C_1(s)$
  in main thread Insert s in Z'_1 `# omp pragma: master`
  synchronization barrier `# omp pragma: barrier`
  split among threads, `# omp pragma: for`
  foreach $t \in Z_1$ *such that* $t \neq s$ *and* $C_1(t) > C_1(s)$ **do**
    $cst \leftarrow \max\{C_1(s), d(s,t)\}$
    if $cst < C_1(t)$ **then**
      If $C_1(t) \neq +\infty$
      $P_1(t) \leftarrow s$, $L_1(t) \leftarrow L_1(s)$, $C_1(t) \leftarrow cst$
      Insert or Update cost for t in Q `# omp pragma: master`
return classifier $[P_1, C_1, L_1, Z'_1]$

Algorithm 3: POPF-PQ Training Algorithm

This last idea can be applied to Prim's algorithm as well. Algorithm 4, which we refer to as simply **POPF**, summarizes all these ideas. As in the previous algorithm, OpenMP pragmas used in the implementation are included as comments. The same approach from the previous algorithm, regarding launching of threads, was used.

Input: A λ-labeled training set $\mathcal{Z}_1$, and number m of threads.
Output: Optimum path forest P_1, cost map C_1, label map L_1 and cost-ordered set Z'_1

$Z'_1 \leftarrow \emptyset$
$\mathcal{S}^* \rightarrow SelectPrototypes(\mathcal{Z}_1)$
foreach $s \in Z_1 \setminus S$ **do**
  assign $C_1(s) \leftarrow +\infty$
foreach $s \in S$ **do**
  $C_1(s) \leftarrow 0$, $P_1(s) \leftarrow nil$, $L_1(s) \leftarrow \lambda(s)$
s $\leftarrow$ any element from $\mathcal{S}^*$
Insert s in Z'_1
in parallel for threads $i = 1, 2, \ldots, m$ `# omp pragma: parallel`
while $s \neq nil$ **do**
  $s_i \leftarrow nil$
  split among threads, `# omp pragma: for`
  foreach $t \in Z_1$ *where* $t \neq s$ **do**
    if $C_1(t) > C_1(s)$ **then**
      $cst \leftarrow \max\{C_1(s), d(s,t)\}$
      if $cst < C_1(t)$ **then** $P_1(t) \leftarrow s$, $L_1(t) \leftarrow L_1(s)$, $C_1(t) \leftarrow cst$
    if $s_i = nil$ or $C_1(t) < C_1(s_i)$ **then** $s_i \leftarrow t$
  in main thread s $\leftarrow \arg\min_{s_i} C(s_i)$ **in**
  in main thread Insert s in Z'_1 `# omp pragma: master`
  synchronization barrier `# omp pragma: barrier`
return classifier $[P_1, C_1, L_1, Z'_1]$

Algorithm 4: POPF Training Algorithm

3.3 Vectorization of distance calculations

Recall that OPF uses as edge weights the distance between the feature vectors of the objects. In general, distance calculations are expensive operations due to the use of floating point operations. For instance, LibOPF (J. Papa et al., 2014) uses by default the logarithm of the euclidean distance between the feature vectors, namely:

$$d(u,v) = D \cdot \log(1 + ||\vec{u} - \vec{v}||^2) = D \cdot \log\left(1 + \sum_{i=1}^{f}(u_i - v_i)^2\right), \tag{3.1}$$

where D is a constant, f is the number of features and u_i, v_i are the i-th component of vectors $\vec{u}$ and $\vec{v}$, respectively. The summation at the end of Equation 3.1 is a vector operation, meaning that it can be implemented with vector or SIMD instructions that some processors provide.

Thus, distance calculations during OPF training can take advantage of this technique. Even though we are focusing on this particular distance function, any distance function usually has a component that can be vectorized as well.

Vectorization can be easily implemented with the Advanced Vector Extensions (AVX) 2 Instruction Set for x86_64 architectures.

Chapter 4

Spatial Indexing for the OPF Classification

When several samples need to be classified at the same time, a simple approach is to classify all of them in parallel by partitioning the set of samples. The OPF classifier, once built, can be accessed concurrently by Algorithm 2 in a read-only fashion. Thus, there are no critical sections to concern about for implementing this approach.

Nevertheless, in this chapter we present new techniques to reduce the number of distance calculations while classifying a sample. We effectively reduce the running time of the classification for each sample by replacing the inner loop of Algorithm 2, as presented in (J. P. Papa et al., 2010), with a query to a data structure built on top of the trained optimum-path forest.

The following sections assume the use of $f_{\max}$ as path cost function for building the OPF and classifying new samples.

4.1 Spatial Indexing of the OPF

Recall that, during classification, for each sample $\mathbf{t}$ we look for the node $\mathbf{s}^*$ in the OPF which offers the least $C(\mathbf{t})$ to the new sample. Then, node $\mathbf{t}$ is assigned label $L(\mathbf{s}^*)$ as calculated by the learning process. This is, for each new sample, we try to find a sample in the OPF which offers the lowest path cost.

This process resembles the search for the closest neighbor for a feature vector in a feature space, also known as proximity query. Several techniques have been proposed to solve this problem (Gaede & Günther, 1998; Chávez, Navarro, Baeza-Yates, & Marroquín, 2001; Indyk & Motwani, 1998), which involve the creation of a data structure or spatial index. It is worth noting that, in fact, spatial indexes have being successfully used for implementing k-Nearest Neighbor classifiers.

Metric Access Methods (Chávez et al., 2001) are particularly suited for proximity queries when dealing with a metric space, i.e. a space for which a metric distance is

defined. These methods exploit the properties of a metric distance, such as the triangle inequality, to build an index and be able to discard elements quickly when doing a query.

Recall that edges in the OPF are weighted by the distance between their adjacent nodes in the feature space. However, once the optimum-path forest is built, there is no clear distance defined for two nodes in terms of the forest itself. If two nodes are not in the same tree or have no ancestor-descendant relationship, a distance can not be established for the path cost function.

4.1.1 OPF-distance

To overcome this issue, let's define the function $\delta : \mathcal{Z} \times \mathcal{Z}_1 \to \mathbb{R}$, which we name OPF-distance, as:

$$\delta(\mathbf{u}, \mathbf{v}) = \max\{d(\mathbf{u}, \mathbf{v}), C(\mathbf{v})\} \tag{4.1}$$

where $\mathbf{u} \in \mathcal{Z}$, $\mathbf{v} \in \mathcal{Z}_1$, d is the distance defined for space $\mathcal{Z}$ and C is the cost map as calculated by Algorithm 1. Note that the definition of δ implies:

$$d(\mathbf{u}, \mathbf{v}) \leq \delta(\mathbf{u}, \mathbf{v}) \tag{4.2}$$
$$C(\mathbf{v}) \leq \delta(\mathbf{u}, \mathbf{v}) \tag{4.3}$$

Note that δ is not commutative and it is undefined if $\mathbf{v} \notin \mathcal{Z}_1$. Also, observe that Equation 2.3 can be rewritten as:

$$C(\mathbf{t}) = \delta(\mathbf{t}, \mathbf{s}^*) \tag{4.4}$$

.

The classification of a sample can now be redefined as *nearest neighbor* search, but using the OPF-distance δ instead of just the distance d. However, even for metric spaces where d satisfies the triangle inequality, we can not guarantee that δ satisfies it too.

4.1.2 OPF Triangle Inequality

As we mentioned before, metric access methods use the triangle inequality of the metric distance to safely discard candidates on a proximity query. The triangle inequality for a distance d in a feature space $\mathcal{X}$ is defined as follows:

$$\forall \mathbf{x}, \mathbf{y}, \mathbf{z} \in \mathcal{X}, d(\mathbf{x}, \mathbf{y}) \leq d(\mathbf{x}, \mathbf{z}) + d(\mathbf{z}, \mathbf{y}) \tag{4.5}$$

On the other hand, the OPF-distance δ introduced earlier is not a metric distance, even if d is. As such, it does not satisfy the triangle inequality. However, it satisfies a couple of inequalities, which we collectively name as OPF Triangle Inequalities.

Lemma 1. *Given $\boldsymbol{p}, \boldsymbol{q}, \boldsymbol{t} \in \mathcal{Z}$, the OPF-distance δ satisfies:*

$$\delta(\boldsymbol{p}, \boldsymbol{q}) \leq d(\boldsymbol{p}, \boldsymbol{t}) + \delta(\boldsymbol{t}, \boldsymbol{q}) \tag{4.6}$$

Lemma 2. *Given $\boldsymbol{p}, \boldsymbol{q}, \boldsymbol{t} \in \mathcal{Z}$, where $C(\boldsymbol{p}) \leq C(\boldsymbol{q})$, the OPF-distance δ satisfies:*

$$\delta(x, p) \leq \delta(p, q) + \delta(x, q) \tag{4.7}$$

See Appendix A.1 for proofs of these lemmas.

4.1.3 The OPF Tree

Using the OPF Triangle Inequality (Lemas 1 and 2) and based on the Vantage-Point Tree (VPT) (Uhlmann, 1991), we now present an index for the OPF, which we name the OPF Tree. We chose to base the OPF Tree on the VPT for the following reasons:

- it has $\mathcal{O}(n \log(n))$ construction time,
- it occupies $\mathcal{O}(n)$ space,
- it is one of the simplest data structures designed for continuous distance functions where set of elements do not change,
- it was designed for main memory.

The design of the OPF Tree comprises two algorithms, one for building and one for querying, which we describe next.

OPF Tree Construction

The OPF tree is built recursively from $\mathcal{Z}_1$, as described in Algorithm 5. Any node $\mathbf{s} \in \mathcal{Z}$ with $C(\mathbf{s}) = 0$ (i.e. a prototype) is chosen as the root. Then, the median M of the set of all OPF-distances is obtained:

$$M = \text{median}\{\delta(\mathbf{s}, \mathbf{u}) / u \in \mathcal{Z}_1 \setminus \{\mathbf{s}\}\} \tag{4.8}$$

Those elements $\mathbf{u}$ such that $\delta(\mathbf{p}, \mathbf{u}) \leq M$ are placed on the left subtree, while those such that $\delta(\mathbf{p}, \mathbf{u}) > M$ are placed on the right subtree. This is illustrated in Figure 4.1a. Then, for each subtree we chose as root any node s' with least cost $s' = \arg\min\{C(s')\}$ and repeat the process until only one node is left. The node with least cost is needed to fulfill the hypotheses for Lemma 2. This requirement leads to nodes always having a cost greater than or equal to their parents' cost.

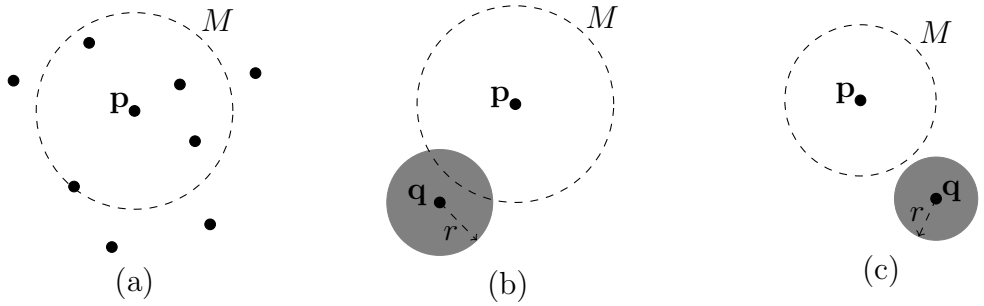


Figure 4.1: (a) Construction of an OPF Tree for the root $\mathbf{p}$: nodes with distance $\delta(\mathbf{p}, \mathbf{u} \leq M$ go to the left subtree and nodes with distance $\delta(\mathbf{p}, \mathbf{u}) > M$ go to the right subtree. (b) Querying an OPF Tree with root $\mathbf{p}$, query sample $\mathbf{q}$ and current radius r where no subtrees can be discarded. (c) Querying an OPF Tree with root $\mathbf{p}$, query sample $\mathbf{q}$ and current radius r where the left subtree can be discarded.

OPF Tree Query

As described in Algorithm 6, to do a query with a sample $\mathbf{t}$, we keep a record of the minimal cost r we have probed for t along the progress of the query. r is set to ∞ at the begining. We start with the root p and calculate $\Delta = \delta(\mathbf{t}, \mathbf{p})$ and update r. If $\Delta - r \leq M$, we explore the left subtree and ignore it otherwise. Simmilarly, if $\Delta + r > M$, we explore the right subtree and ignore it otherwise. Next, we do the same process recursively for each subtree or finish the search if no subtrees are left. Figure 4.1b illustrates an instance where no subtree can be discarded and Figure 4.1c illustrates an instance where the left subtree can be discarded for certain iteration.

At the end of the query algorithm, $C(\mathbf{t}) = r$ and its label $\lambda(\mathbf{t})$ is set to the label of the node $\mathbf{s}^*$ for which $r = \delta(\mathbf{t}, \mathbf{s}^*)$. An additional observation is that we can also drop a whole subtree if its root p has a cost $C(p)$ greater than r, because the cost of its nodes increase as we go further down in the tree.

The OPF Triangle Inequality ensures that we do not miss $\mathbf{s}^*$. All the subtrees not traversed contain elements $\mathbf{q}$ such that $\delta(\mathbf{t}, \mathbf{q}) > r$. And because of Equation 4.4, $\mathbf{s}^* \neq \mathbf{q}$. The following lemmas support this claim (See Appendix A.2 for their proofs).

Lemma 3. *Given $\boldsymbol{p}, \boldsymbol{q}, \boldsymbol{t} \in \mathcal{Z}$, where $d(\boldsymbol{p}, \boldsymbol{t}) + r \leq M$ and $\delta(\boldsymbol{p}, \boldsymbol{q}) > M$, then $\delta(\boldsymbol{t}, \boldsymbol{q}) > r$.*

Lemma 4. *Given $\boldsymbol{p}, \boldsymbol{q}, \boldsymbol{t} \in \mathcal{Z}$, where $C(\boldsymbol{p}) \leq C(\boldsymbol{q})$, $\delta(\boldsymbol{p}, \boldsymbol{q}) \leq M$ and $\delta(\boldsymbol{t}, \boldsymbol{p}) - r > M$, then $\delta(\boldsymbol{t}, \boldsymbol{q}) > r$.*

Depth-First and Priority Order

Recall that, during the algorithm, we keep track of the minimal cost r provided by the visited nodes. It is easy to see that if r reduces quickly, then more branches of the tree are likely to be discarded. Thus, we want to visit the nodes of the OPF Tree in an order that makes this happen.

On the implementation presented on Algorithm 6 we visit the branches of the OPF Tree as they are discovered, in a depth-first search fashion. Alternatively, we can prioritize

the order in which the algorithm expands on a branch. Similarly to the approach seen in Section 2.1.2, we can visit the nodes in the order of increasing cost. In the case of the OPF Tree query algorithm, we use the cost of the root of the subtree. Therefore, when a new subtree is discovered by the algorithm, it is queued with a priority value equal to its root cost. At each point, we select for expansion the subtree with the least cost.

Note that the OPF Tree query with priority order guarantees that the number of comparisons done is not greater than the number of comparisons of Algorithm 2.

Input: The trained OPF classifier, described by a cost map C_1 and cost-ordered set Z'_1

Output: The OPF Tree, rooted at *root* and with each node described by relationships $left\_child$ and $right\_child$, a median *med* and a pointer *snode* to the OPF node.

```
1  Function BuildOPFTree(C_1, Z'_1)
2      root ← new Node;
3      BuildOPFTreeHelper(root, C_1, Z'_1)
4      return root
5  Function BuildOPFTreeHelper(node, C, Z)
6      node.snode ← pop_first(Z)
7      Δ ← {}
8      foreach s ∈ Z do
9          Δ(s) ← max{d(s, node.snode), C(node.snode)}
10     node.med ← median{Δ}
11     Z_l ← ∅
12     Z_r ← ∅
13     foreach s ∈ Z do
14         if Δ(s) <= node.med then
15             Insert s in Z_l
16         else
17             Insert s in Z_r
18     if Z_l ≠ ∅ then
19         node.left_child ← new Node
20         BuildOPFTreeHelper(node.left_child, C, Z_l)
21     if Z_r ≠ ∅ then
22         node.right_child ← new Node
23         BuildOPFTreeHelper(node.right_child, C, Z_r)
```

Algorithm 5: OPF Tree Construction Algorithm

Paralell Algorithms for the OPF Tree

Recall that, while POPF improves the OPF training, the OPF Tree improves the OPF classification. However, both processes, training and classification, are repeatedly applied one after another when performing the learning procedure (Section 2.1.3). Thus, the performance of both algorithms can be compared in this phase. Given the parallel nature of POPF, we also need to parallelize the OPF Tree construction and query in order to take advantage of multithreading and compare both algorithms as the number of threads

Input: A sample t, the trained OPF classifier, described by a cost map C_1 and label map L_1, and the OPF Tree rooted at *root*
Output: The label for the sample t stored in map L_2
1 **Function** *QueryOPFTree(t, C_1, L_1, root)*
2 | $QueryOPFTreeHelper(t, C_1, L_1, root, \infty)$
3 **Function** *QueryOPFTreeHelper(t, C_1, L_1, node, r)*
4 | **if** $C_1(node.snode) >= r$ **then**
5 | | **return**
6 | $\Delta \leftarrow \max\{d(t, node), C_1(node.snode)\}$
7 | **if** $\Delta < r$ **then**
8 | | $r \leftarrow \Delta$
9 | | $L_2(t) \leftarrow L_1(node.snode)$
10 | **if** $node.left\_child \neq nil$ *and* $\Delta - r < tnode.med$ **then**
11 | | $QueryOPFTreeHelper(t, C_1, L1, node.left\_child, r)$
12 | **if** $node.right\_child \neq nil$ *and* $\Delta + r > tnode.med$ **then**
13 | | $QueryOPFTreeHelper(t, C_1, L1, node.right\_child, r)$

Algorithm 6: OPF Tree Query Algorithm

increase.

Parallel OPF Tree Query The OPF Tree query is inherently an iterative process. Some parallelization could be applied when both branches of a particular subtree need to be visited (lines 10-13 of Algorithm 6). However, this parallelization fits more in the Fork-Join model where it is difficult to distribute the load evenly among the threads. Basically, we can not know how many nodes of a branch are going to be visited beforehand. Alternatively, to better fit the SIMD that we are already using for POPF, we can distribute the load by partitioning the set of elements to be classified ($\mathcal{Z}_2$ or $\mathcal{Z}_3$). Then, each thread can classify the samples in each partition.

Parallel OPF Tree Construction The OPF Tree construction has a similar structure that also fits the Fork-Join model. In this case, it is more plausible to distribute the load evenly, but we still need to manage some kind of nested forking. Instead, we look for an oportunity to apply partitioning. The most appropriate candidate is the loop in lines 8-9 in Algorithm 5, which also happens to be the most expensive part of the algorithm due to the $O(n)$ distance calculations, whre n is the number of elements in the subtree. Thus, we apply the partitioning on the set of nodes of each subtree formed during the OPF Tree construction.

As an implementation detail, note that it would be needed to spin the threads once for each subtree. This number is $O(n)$, where n is the size of the classifier, and, given that thread creation is costly, it is not appropriate. Instead, we decided to first make the recursion iterative by adding a stack. Then, we can launch the threads only once at the beginning of the algorithm and reuse them for each subtree's distance calculations. Only the main thread creates nodes in the OPF Tree. This approach to thread creation is similar to the one used in both versions of POPF.

Chapter 5

Results

In order to compare the efficiency of approaches proposed in Chapter 3 and Chapter 4, we have conducted several experiments on three different architectures and using seven public datasets with various sizes and number of features. We measured not only the overall speedup, but also accounted for number of distance calculations, when appropriate.

5.1 Datasets

All datasets used in this work were taken from the UCI Machine Learning Repository (Lichman, 2013). We intentionally chose datasets with numeric features (to avoid preprocesing, which is out of the scope of this work) and with sizes of different orders of magnitude, in order to better describe the scalability of our approach.

Table 5.1 presents the datasets used, along with its sizes, number of classes and number of features. Each dataset was split in the three groups $\mathcal{Z}_1$, $\mathcal{Z}_2$ and $\mathcal{Z}_3$, the training, evaluation and testing sets, respectively. For almost all datasets, we chose a partition of 20%, 40% and 40% of the elements for these subsets. The exception was the dataset with biggest size: `SkinSeg` dataset, for which we used a partition of 10%, 20% and 70%.

Table 5.1: Description of the datasets used in the experiments

Dataset	**instances**	**features**	**classes**
Letter Recognition (Lichman, 2013)	20,000	16	26
Statlog (Shuttle) (Lichman, 2013)	43,500	9	7
Dataset for **S**ensorless **D**rive **D**iagnosis (Lichman, 2013)	58509	49	11
MiniBooNE Particle Identification (Lichman, 2013)	130,064	49	2
Skin Segmentation (Lichman, 2013)	245,057	3	2
Pen-Based Recognition of Handwritten **Digits** (Lichman, 2013)	7,494	16	10
MAGIC Gamma Telescope (Lichman, 2013)	19020	11	2

Table 5.2: Machine/System configuration used in the experiments for the training algorithms

Component	Machine 1	Machine 2
Processor	Intel® Xeon™ E5-2620 v3 @ 2.4GHz	AMD® Opteron™ Processor 6272 @ 2.1GHz
Sockets	2	4
Cores per socket	6	8
Threads per core	2	2
SIMD technology	AVX2	AVX1
L1 instruction cache	32KiB 8-way set associative	64KiB 2-way set associative (shared)
L1 data cache	32KiB 8-way set associative	16KiB 4-way set associative
L2 (unified)	256KiB 8-way set associative	8x2MiB 16-way set associative
L3 (unified+shared)	15MiB 20-way set associative	12MiB up to 64-way set associative
RAM	56GiB DDR4-1866, 3*16GiB + 1*8GiB DIMMS	64GiB DDR3-1066, 4*16GiB DIMMS
Max bandwidth	59 GiB/s	51.2 GiB/s
Operating System	Linux kernel 4.4.0 (64-bit)	Linux kernel 4.9.1 (64-bit)
Compiler	gcc (GCC) 5.2.1	gcc (GCC) 5.2.1
Optimization flag	-O3	-O3

5.2 The Test Environments

For assessing the performance of our parallel training algorithms, we conducted our experiments in two different machines with different levels of parallelism. These machines' characteristics are shown in Table 5.2.

As one can observe, Machine 1 has a Xeon Intel processor whereas Machine 2 uses an AMD Opteron. The access to memory in both machines is not uniform (NUMA), that is, if a memory position accessed by a core in the first socket is present in the DRAM slot assigned to the second socket, the latency will be higher than if the memory was present in the first socket. The Linux kernel version, for both machines, automatically deals with distributing and migrating memory pages around in order to reduce the NUMA impact. This feature is commonly known as AutoNUMA (*AutoNUMA: the other approach to NUMA scheduling*, 2012).

On the other hand, for assessing the performance of the spatial indexing for the OPF classification, we used a commodity hardware machine with an Intel Core i7 processor. This is because the focus of the technique is to reduce the number of distance calculations performed, instead of the scalability of the algorithm when varying the number of threads. This machine's characteristics are listed in Table 5.3.

5.3 Methodology and Results for the Training Algorithm

In Chapter 3 we presented two parallel algorithms for the OPF training, which we called **POPF-PQ** (that preserves the priority queue from the serial algorithm) and **POPF** (that

Table 5.3: Machine/System configuration used in the experiments for the classification algorithms

Component	Machine 3
Processor	Intel® Core™ i7-4510U @ 2.0GHz
Sockets	1
Cores per socket	2
Threads per core	2
SIMD technology	AVX2
L3 cache	4MiB
RAM	12GiB DDR3
Operating System	Linux kernel 4.13.12 (64-bit)
Compiler	gcc (GCC) 7.2.0
Optimization flag	-O3

uses an array and a linear search). Additionally, we discussed the use of vectorization to speed up the distance calculations.

We now present comparisons among all versions of the training algorithm discussed. Each experiment was executed 5 times, and the results discussed in the next sections represent the averaged execution time (Section 5.3.1) and speedup (Section 5.3.2) for the learning process. In all cases, the confidence interval (95%) was calculated and is shown as error bars in the graphs reporting the execution time.

5.3.1 Execution time

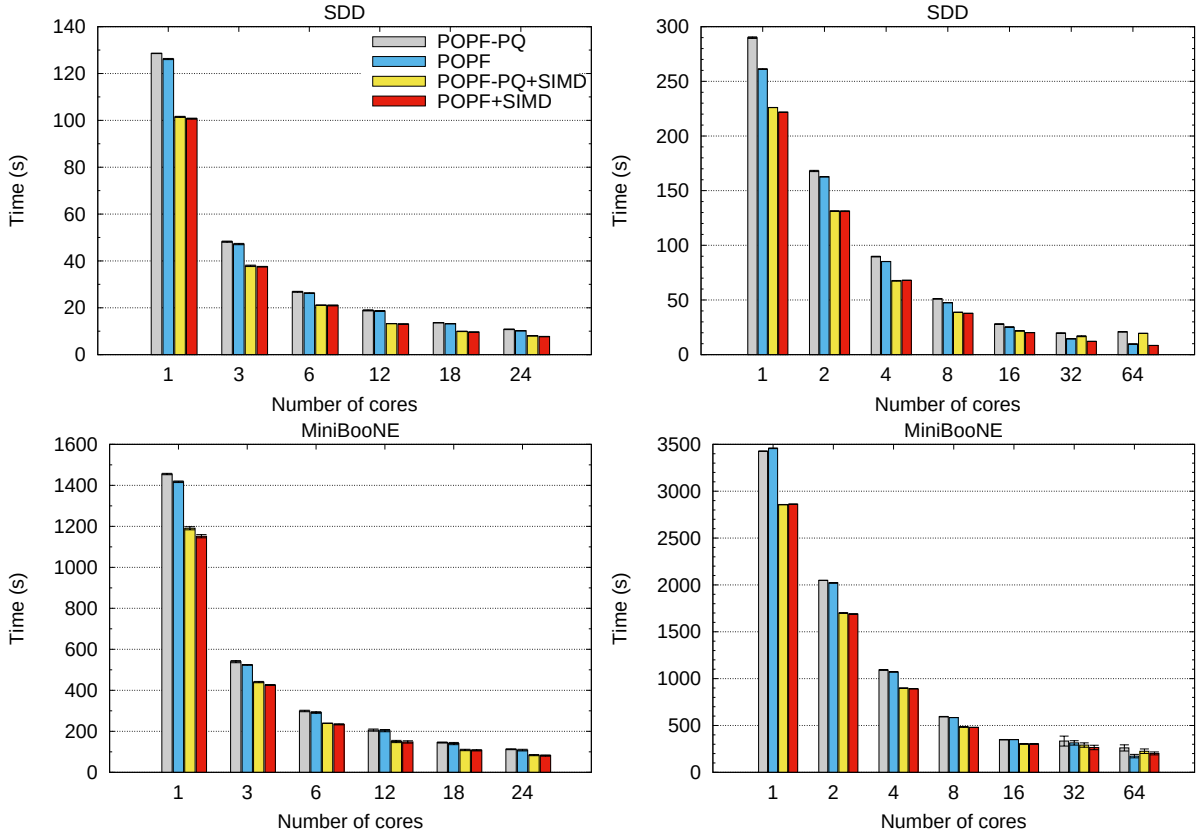
In order to better present the results, the discussion is divided according to the effectiveness of the vectorization technique. We start with the SIMD-friendly datasets, in other words, those that were processed faster with the vectorized distance computation. Next, we present those that did not benefit from the use of SIMD instructions (SIMD-unfriendly). Finally, the results for the dataset oblivious to vectorization is discussed (SIMD-oblivious).

SIMD-friendly

Figure 5.1 shows the execution time (y-axis) of both OPF implementations (`POPF-PQ` and `POPF`) when processing `SDD` and `MiniBooNE` for each number of threads (x-axis). Figure 5.1 also shows the results with the vectorized distance computation (`POPF-PQ+SIMD` and `POPF+SIMD`). Figure 5.1(a) and 5.1(b) stand for the results considering the Xeon™ (`Machine 1`) and Opteron™ (`Machine 2`) machines, respectively.

One can observe that both `POPF-PQ` and `POPF` are able to run faster as the number of threads increase. `POPF` was slightly faster than `POPF-PQ`, particularly on `Machine 2` for `SDD`. This result is due to the fact that the NUMA effect is stronger on `Machine 2` than `Machine 1`. By eliminating the shared priority queue, `POPF` exploits data locality since each thread only changes a private section of the list were the samples are kept.

Figure 5.1 shows that the execution time is significantly reduced by vectorizing the



(a) Machine 1 (b) Machine 2

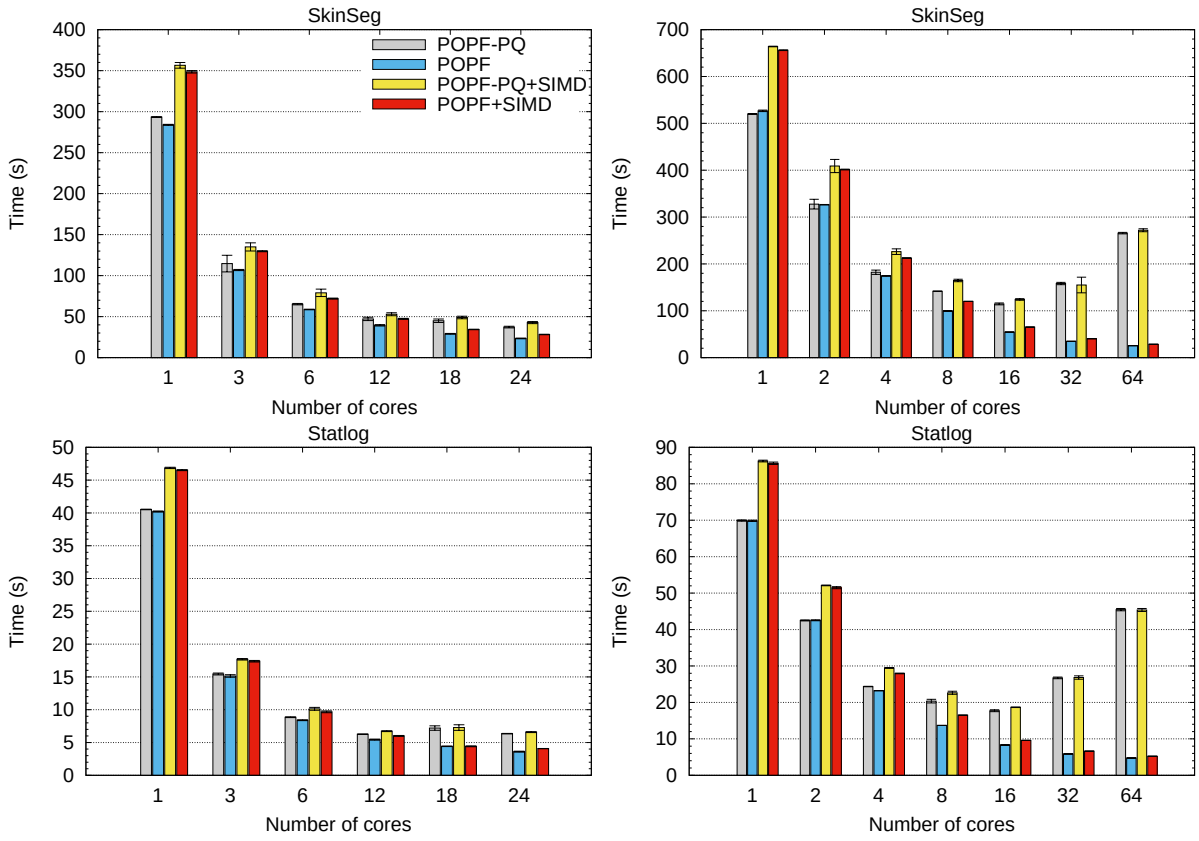
Figure 5.1: Execution time for the training algorithms on the SIMD-friendly datasets

distance computation on both machines. Particularly, Figure 5.1 depicts the performance gains with 1 thread are up to 26.7% on `Machine 1` and 28.8% on `Machine 2` for `SDD`. Such good results were possible because both `SDD` and `MiniBooNE` have a number of features that is a *large* multiple of both machine processor's vector unit. The performance gap between `POPF-PQ` and `POPF` is not greater due to the distance computation. Even with vectorization, such procedure still dominates the learning time for both `SDD` and `MiniBooNE`.

SIMD-unfriendly

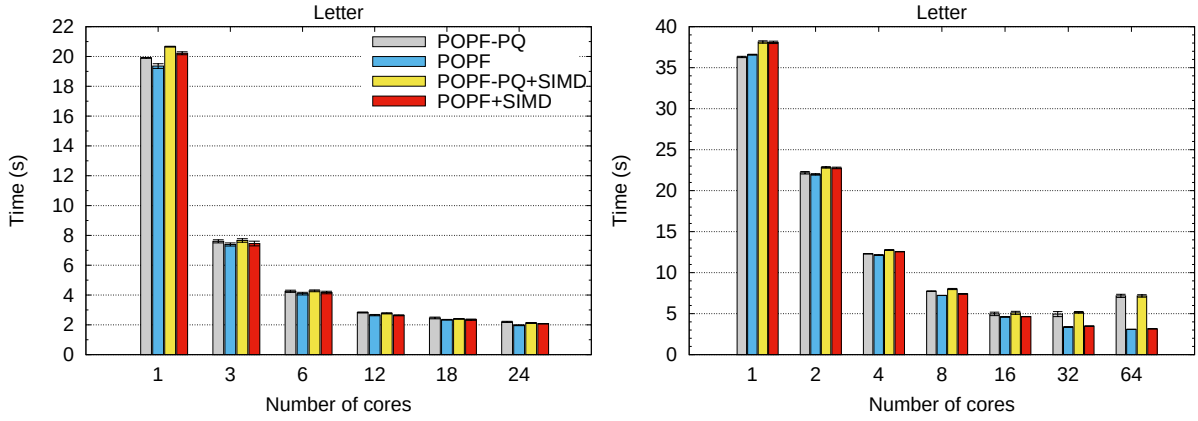
Figure 5.2 shows the execution time results for `SkinSeg` and `Statlog`. As one can observe, `POPF-PQ` and `POPF` are both able to run faster as the number of threads increase. However, with a high thread count, the NUMA effects severely degenerates `POPF-PQ` performance. Such degradation is not observed in `POPF` because it keeps data movement to a minimum by storing each sample candidate in a thread-private location. Indeed, `POPF-PQ` takes almost 60% and 74% longer than `POPF` on `Machine 1` for `SkinSeg` and `Statlog`, respectively, with 24 threads. On `Machine 2`, the degeneration is even worse, since `POPF-PQ` is over 10 and 9 times slower than `POPF` when learning `SkinSeg` and `Statlog`, respectively, with 64 threads.

Figure 5.2 also shows that vectorizing the distance computation can negatively im-



(a) Machine 1 (b) Machine 2

Figure 5.2: Execution time for the training algorithms on the SIMD-unfriendly datasets



(a) Machine 1 (b) Machine 2

Figure 5.3: Execution time for the training algorithms on the SIMD-oblivious datasets

pact performance. Indeed, both `POPF-PQ+SIMD` and `POPF+SIMD` performed worse than their non-vectorized versions due to the overhead of moving *small* feature vectors to the processors' vector unit and sending them back to the scalar units.

SIMD-oblivious

As Figure 5.3 shows, vectorizing the distance computation leads to a negligible performance impact in the learning time of the `Letter` dataset. The performance gains achieved with the data parallelism of SIMD instructions are not enough to trade off the overhead of moving data between the scalar and vector units but, in this case (a feature size of 16), it did not degrade performance either. Similar to both `SkinSeg` and `Statlog` results, `POPF` outperforms `POPF-PQ` considering the `Letter` dataset with higher thread counts on `Machine 2` due to the minimization of the NUMA effect provided by that approach. Indeed, `POPF` is over 2.3x faster than `POPF-PQ` on `Machine 2` with 64 threads.

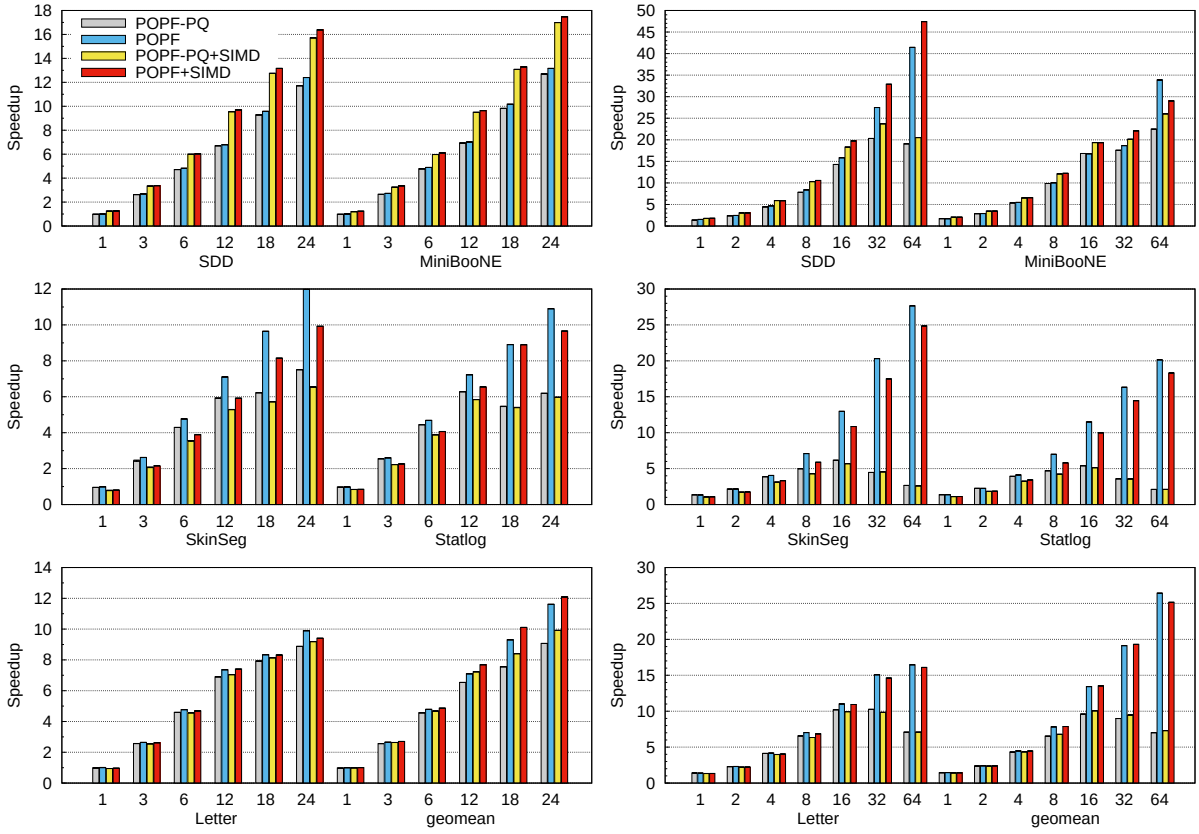
5.3.2 Speedup

When designing parallel programs, it is important to measure how much gain in speed we are obtaining as we increase the number w of workers or threads. In this section, we measure the speedup of our programs, as defined in (Pacheco, 2011):

$$S = \frac{T_{serial}}{T_{parallel}}, \tag{5.1}$$

where T_{serial} is the serial running time and $T_{parallel}$ is the parallel running time. Ideally, we want the speedup would be $S = w$. However, due to the overhead produced by thread creation, deletion and synchronization, it would become an smaller fraction of the ideal.

Figure 5.4 shows the speedup results (y-axis) for both parallel implementations of OPF with (`POPF-PQ+SIMD` and `POPF+SIMD`) and without (`POPF-PQ` and `POPF`) vectorization. For each dataset, the experiments were executed varying the number of threads (x-axis)



(a) Machine 1 (b) Machine 2

Figure 5.4: Speedup results for the training algorithms on the Xeon™ (a) and Opteron™ (b) machines

between 1 and 24 on `Machine 1`, and between 1 and 64 on `Machine 2`. The cluster of bars labeled `geomean` represents the geometric mean of the speedups achieved for each dataset.

As Figure 5.4 shows, both implementations achieved their best speedup results with `SDD` and `MiniBooNE`. Particularly, with the distance computation vectorized, both implementations were over 32% faster than their scalar counterparts on `Machine 1` with 24 threads. In addition, `POPF+SIMD` performed over 14% better than `POPF` with `SDD` on `Machine 2` with 64 threads. The use of SIMD instructions was not as effective on `Machine 2` as on `Machine 1` mainly because pairs of Opteron™ cores share the floating point unit, so only one of them can use it at a time.

The results for `SkinSeg` and `Statlog` show how severely NUMA impacts the performance of algorithms with frequent synchronization needs (e.g. `POPF-PQ`). Figure 5.4 shows that `POPF-PQ` is up to 60% (`SkinSeg`) and 77% (`Statlog`) slower than `POPF` on `Machine 1` with 24 threads. The performance becomes worse on `Machine 2` which has more sockets. Indeed, `POPF-PQ` is over 10x (`SkinSeg`) and 9x (`Statlog`) slower than `POPF` on `Machine 2` with 64 threads. By comparing the average results on both machines it is clear that, as the number of sockets increase, the more important it is to keep communication/synchronization to a minimum.

5.4 Methodology and Results for the Classification Algorithm

In Chapter 4 we presented the OPF Tree, a data structure that aims to reduce the number of comparisons of new samples with OPF elements during classification. We now measure the performance of the OPF Tree in comparison to the original classification algorithm.

Given the nature of the algorithms – which discard candidates from the OPF during the classification of a sample – we measure not just the overall running time (Section 5.4.2), but also the number of distance calculations performed (Section 5.4.1).

5.4.1 Number of Distance Calculations

Originally, the OPF classification algorithm tried to extend paths from every node of the classifier in the search for an optimum path for the new sample (J. P. Papa et al., 2009). A later optimization allowed to discard some nodes by means of a sorted list of nodes by cost (J. P. Papa et al., 2010). This is important because extending a path requires a distance calculation, which is computationally intensive. Thus, the OPF tree aims to discard a greater number of nodes by means of exploiting properties of metric distances.

In this section we compare the average number of distance calculations that are performed for each algorithm when classifying a set of samples. This is shown not just for the testing phase, but also for the learning phase. Recall that this process repeatedly performs the classification of an evaluation set to improve the precision of the classifier. The average numbers of distance calculations are presented as a percentage of the size of the OPF classifier, which is also the size of the OPF Tree.

In Figures 5.5, 5.6 and 5.7 we denote the original classification algorithm by simply `OPF`, and our techniques by `OPF-tree` and `OPF-tree-`ρ, respectively. The former refers to the variant that runs in a depth-first order fashion and the later refers to the variant that runs in priority order based on the cost C of the elements of the OPF. The x-axis represents the iteration number for the learning algorithm (where classification is performed for $\mathcal{Z}_2$). The latest iteration represents the testing phase (where classification is performed for $\mathcal{Z}_3$).

In order to better analyze the results, we have divided the datasets in three groups according to how performant the OPF Tree was when compared to the original OPF classification algorithm. At the end of this section, we discuss the reason for these performance differences.

Datasets where OPF Tree was highly performant

Figure 5.5 contains the datasets on which `OPF-tree` and `OPF-tree-`ρ greatly outperform `OPF` in number of distances calculations. Examining only the testing phase, the OPF Tree performed just 2% (for depth-first order) and 4.4% (for priority order) of the number of comparisons of the original algorithm on the `SkinSeg` dataset. On the `Statlog` dataset,

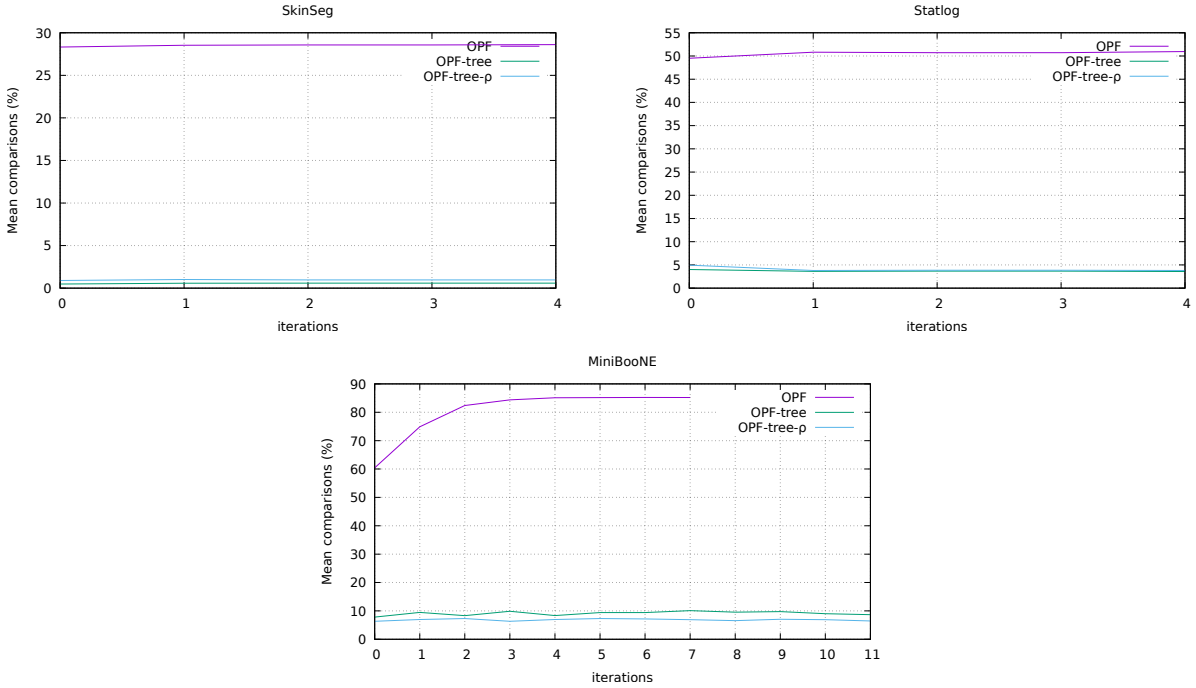


Figure 5.5: Number of Distance Calculations for the classification algorithms on Low Intrinsic Dimensionality Datasets

the OPF Tree performed 7% (for depth-first order) and 7.5% (for priority order) of the number of comparisons of the original algorithm. Finally, on the `MiniBooNE` dataset, the OPF Tree performed 10% (for depth-first order) and 7.6% (for priority order) of the number of comparissons of the original algorithm. However, in this case the `OPF` performed a lower number of iterations in the learning phase.

As seen, `OPF-tree` performed better than `OPF-tree-`ρ on datasets `SkinSeg` and `Statlog`, which happen to be the datasets with lower number of features. The `OPF-tree-`ρ was the best technique for `MiniBooNE` dataset.

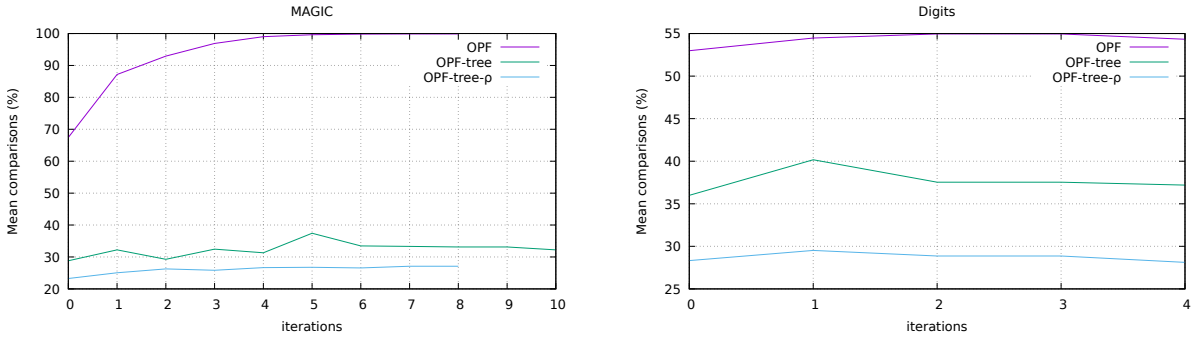
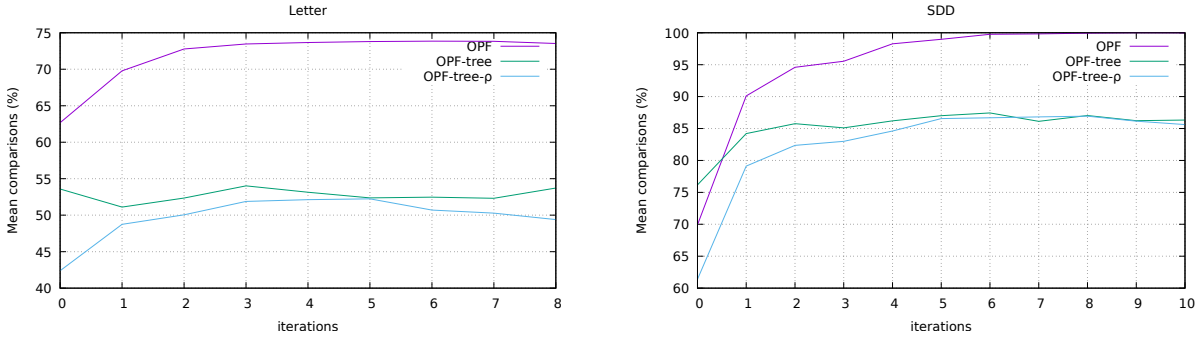


Figure 5.6: Number of Distance Calculations for the classification algorithms on Medium Intrinsic Dimensionality Datasets



(a) (b)

Figure 5.7: Number of Distance Calculations for the classification algorithms on High Intrinsic Dimensionality Datasets

Datasets where OPF Tree was reasonably performant

Figure 5.6 contains the datasets on which `OPF-tree` and `OPF-tree-`ρ performed reasonably better than `OPF` regarding the number of distance calculations. Examining only the testing phase, the OPF Tree perfomed 32% (for depth-first order) and 26% (for priority order) of the number of comparisons of the original algorithm on the `MAGIC` dataset. On the `Digits` dataset, the OPF Tree performed 68% (for depth-first order) and 52% (for priority order) of the number of comparisons of the original algorithm.

In this case, `OPF-tree-`ρ performed better than `OPF-tree` on both datasets, which have around the same number of number of features.

Datasets where OPF Tree was somewhat performant

Lastly, Figure 5.7 contains the datasets on which `OPF-tree` and `OPF-tree-`ρ performed just slightly better than the original `OPF`. Examining only the testing phase, the OPF tree did 73% (for depth-first order) and 67% (for priority order) of the number of comparisons of the original algorithm on the `Letter` dataset. On the `SDD` dataset, the OPF tree did 86.3% (for depth-first order) and 85.6% (for priority order) of the number of comparisons of the original algorithm.

In this last set of datasets, the `OPF-tree-`ρ also performed better than `OPF-tree`, with a very small margin in the case of `SDD` dataset.

OPF Tree and the Intrinsic Dimensionality

In the past paragraphs, we have seen how the OPF Tree performed very differently with different datasets. On a first attempt to explain this, one could relate the performance to the number of features of the datasets. For instance, the dataset on which the OPF Tree performed best was `SkinSeg` and the dataset on which it performed worst was `MiniBooNE`, which happen to be the datasets with lowest number of features and highest number of

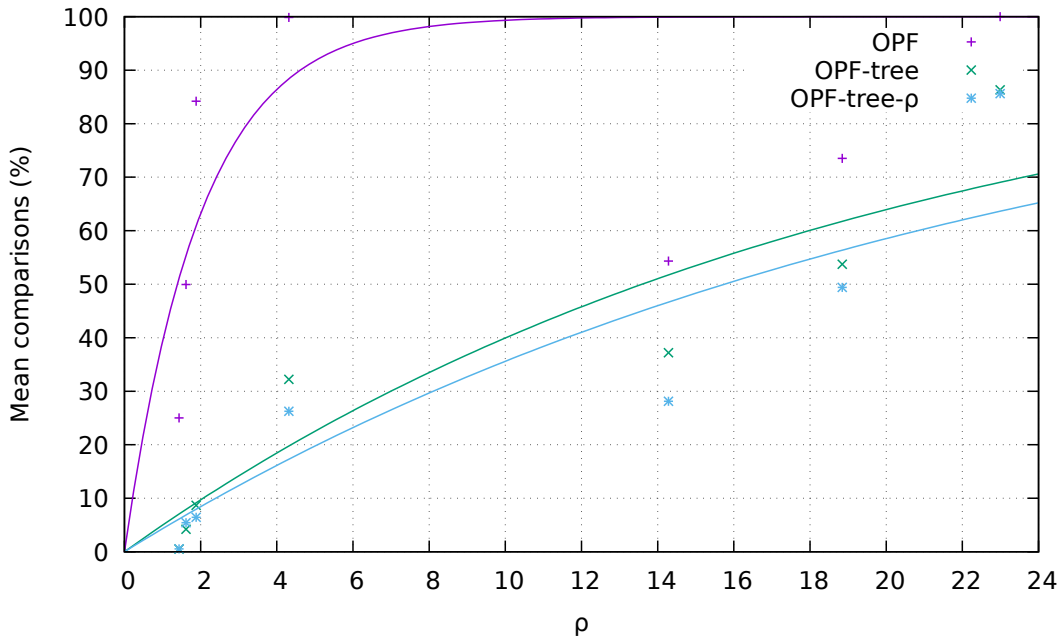


Figure 5.8: Intrinsic dimensionality of datasets vs. the number of distance calculations on classification algorithms

features, respectively. However, both `MiniBooNE` and `SDD` datasets have the same number of features but the performance on them was very different.

Chavez et. al. relate the performance of the metric access methods to the so called intrinsic dimensionality. Given a metric space $\mathcal{X}$, the intrinsic dimensionality ρ is defined as follows (Chávez et al., 2001):

$$\rho = \frac{\mu^2}{2\sigma^6} \tag{5.2}$$

where μ and ρ^2 are, respectively, the mean and variance of the histogram of distances of the space. They showed that as the intrinsic dimensionality of the space increases, the performance of the multidimensional access methods degrades.

We decided to relate the performance of the OPF Tree to the intrinsic dimensionality, too. Figure 5.8 shows the relationship between the intrinsic dimensionality of the datasets and the performance of the classification algorithms (only testing phase considered). The continuous graphs are a least square approximation of the data with functions of the form $1 - \exp(-c \cdot x)$, where c is a constant and x is the intrinsic dimensionality.

This family of functions converge to 1 in the infinity, allowing us to express the fact that, as the intrinsic dimensionality increases, no nodes can be skipped by the algorithms. We see that both versions of the OPF Tree are comparable, but with the priority order version skipping slightly more nodes than the depth-first order version. Both outperform the original OPF algorithm.

5.4.2 Execution Time

In Section 4.1.3 we also introduced the parallel version of the OPF Tree construction and querying. We now assess its running time performance, including building and classification, by comparing `POPF-tree` and `POPF-tree-`ρ against `POPF` (non-vectorized version). Recall that `POPF` was our most performant parallel training algorithm, as we have seen in Section 5.3. We compare both the learning (which repeatedly performs classifications) and evaluation times. Note that the `P` in `POPF-tree` and `POPF-tree-`ρ stands for their parallel nature.

Just like we did when discussing the number of distances calculations, we split the datasets in three sets according to how well the OPF Tree performed on them. We start with the OPF Tree-friendly datasets, on which the `POPF-tree` and `POPF-tree-`ρ outperformed `POPF`. Next, we present the OPF Tree-unfriendly datasets, on which `POPF` outperformed `POPF-tree` and `POPF-tree-`ρ. Lastly, we present the results for the OPF Tree-oblivious datasets, on which the algorithms performed comparably. We used three degrees of parallellism: 1, 2 and 4 threads. Each experiment was performed 5 times. In all cases, the standard deviation was calculated and is shown as error bars in the graphs reporting the execution time.

5.4.3 OPF Tree-friendly

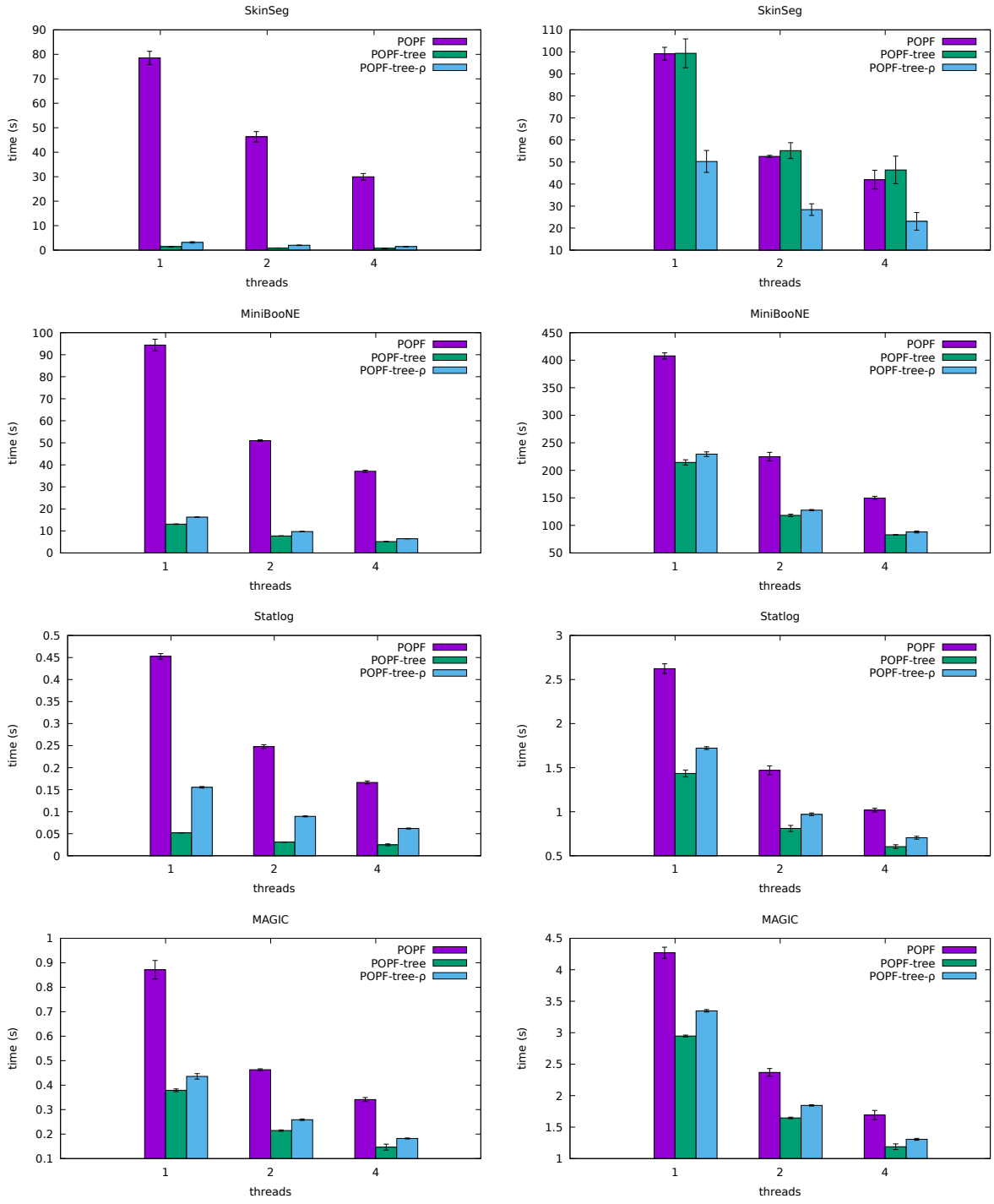
Figure 5.9 shows the running times for the datasets where the `POPF-tree` and `POPF-tree-`ρ vastly outperformed `POPF`. In particular, the best results were obtained for the `SkinSeg` dataset where, for the greatest degree of parallelism, the `POPF-tree` was 41 times faster and the `POPF-tree-`ρ was 21 times faster than `POPF` in the classification phase. For the `MiniBooNE` dataset, which has the highest number of features, the `OPF-tree` was 7.1 times faster and the `OPF-tree-`ρ was 5.7 times faster than `POPF`. Next, for the `Statlog` dataset, the `OPF-tree` was 6.7 times faster and the `POPF-tree-`ρ was 2.7 times faster than `POPF`. Lastly, for the `MAGIC` dataset, the `POPF-tree` was 2.3 times faster and the `OPF-tree-`ρ was 1.9 times faster than `POPF`.

It is worth noticing that, in all cases the `POPF-tree` performed best. This is also true for the datasets where the `POPF-tree-`ρ performs a lower amount of distance calculations. This can be attributed to the internal costs of maintaining the priority queue.

For all of the cases, the learning phase did not take longer for the `POPF-tree` and `POPF-tree-`ρ when compared to `POPF`. The best representative was the `MiniBooNE` dataset where the `OPF-tree` was 1.8 times faster and `POPF-tree-`ρ was 1.7 times faster than `POPF` when using 4 threads. In the case of `SkinSeg` dataset, `OPF-tree` performed roughly equally to `POPF`.

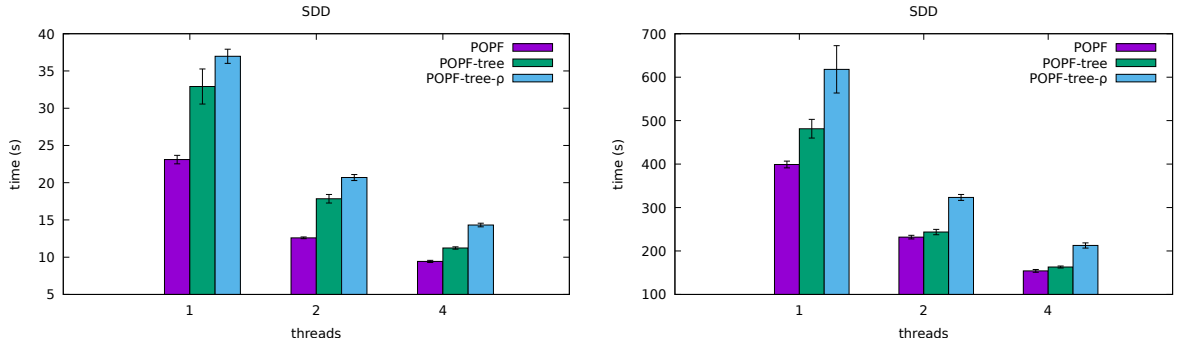
5.4.4 OPF Tree-unfriendly

On the `SDD` dataset, the OPF Tree did not perform well, as can be seen in Figure 5.10. The `POPF` performed 1.2 times faster than `POPF-tree` and 1.5 times faster than `POPF-tree-`ρ



(a) Evaluation (b) Learning

Figure 5.9: Execution times for the classification algorithms on the OPF Tree-friendly datasets



(a) Evaluation (b) Learning

Figure 5.10: Execution times for the classification algorithms on the OPF Tree-unfriendly datasets

on the evaluation phase. On the learning phase, `POPF` performed equally to `POPF-tree` and 1.4 times faster than `POPF-tree-`ρ.

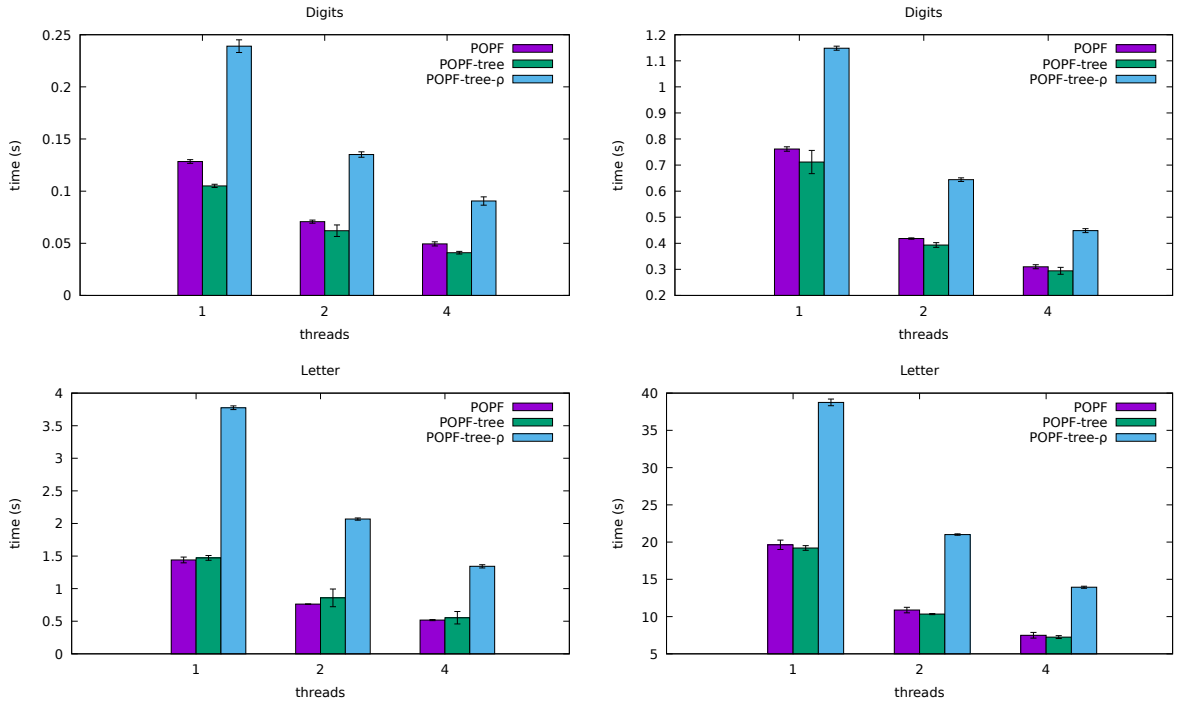
In this case, the lesser number of distance calculations was not enough to account for the OPF Tree building time, for both learning and testing phases.

5.4.5 OPF Tree-oblivious

There were two datasets on which the OPF Tree time performance did not have a significant difference from that of the `POPF`. The graphs can be seen in Figure 5.11. In the `Digits` dataset, `POPF-tree` was 1.2 faster than `POPF`, but `POPF` was 1.8 times faster than `POPF-tree-`ρ. In the `Letter` dataset, the `POPF` performed equally to `POPF-tree` and 2.6 times faster than `POPF-tree-`ρ.

Regarding the learning phase, for the `Digits` dataset, the `POPF` performed equally to `POPF-tree` and 1.4 times faster than `POPF-tree-`ρ. In the `Letter` dataset, the `POPF` also performed equally to `POPF-tree` and 1.9 times faster than `POPF-tree-`ρ.

The OPF Tree with priority order was the clear loser on these datasets, both in the learning and training phase. On the other hand, the difference in performance between the OPF Tree with depth-first order and `POPF` was not significant.



(a) Evaluation (b) Learning

Figure 5.11: Execution times for the classification algorithms on the OPF Tree-oblivious datasets

Chapter 6

Conclusions

This work has two main contributions for the OPF supervised classifier: a coarse-grained parallelization for the OPF training and a new classification algorithm based on a novel spatial index data structure. Both techniques have been proved to effectively improve the running time of all phases of the OPF classifier.

We presented two variants of a parallel OPF training and we demonstrated their efficiency concerning classification tasks. The POPF algorithms, as we named them, are based on three important observations: (i) the optimum-path computation process for each training sample is independent to each other; (ii) the fully connected training graph allows us to perform a data-partition scheme for the nodes of the graph; and (iii) the computation of the MST during the training phase can also be performed in parallel. We also used the second observation to replace the priority queue from the serial algorithm with a parallel linear search on an array without introducing more computational complexity. Our parallel algorithms scaled well as the number of threads increased.

We have also seen that most of the burden of the algorithms is in the distance calculations. We noted that they can be speed up by means of vectorized operations that some processors provide. This allowed us to obtain speedups that doubled the unvectorized versions of the algorithms.

We also introduced the OPF Tree, an spatial data structure that indexes the OPF for faster classifications. For building the OPF Tree, we defined the OPF distance and the OPF triangle inequality, similar to the metric distance and triangle inequality, respectively. These mathematical tools allow the construction of spatial indexes for proximity queries. Similarly, we used the OPF distance and the OPF triangle inequality to build a novel data structure for obtaining the optimum path on the OPF for a query sample.

The performance of the OPF Tree varied for our testing datasets. We were able to establish a relationship between the intrinsic dimensionality of the dataset and the performance of the data structure for classifying it. As a result, we were able to classify samples up to 41 times faster than POPF. On the cases where the OPF tree did not perform as good, it was at least as fast as POPF.

We presented two versions of the OPF Tree query, which vary on the order in which

the nodes of the tree are visited. A priority-based ordering obtained better theoretical results, on which a lesser number of distance calculations are performed. However, on practice, the depth-first order had a better running time, because it does not have the burden of maintaining a priority queue.

Thus, POPF and the OPF Tree allow to perform classification of very large datasets when timing restrictions are present, and they bring closer the possibility of performing nearly real-time classification for reasonable sized-datasets even on a single computer or mobile device.

Appendices

Appendix A

Proofs for the OPF Spatial Indexing

A.1 OPF Triangle Inequality

Lemma 1. *Given $\boldsymbol{p}, \boldsymbol{q}, \boldsymbol{t} \in \mathcal{Z}$, the OPF-distance δ satisfies:*

$$\delta(\boldsymbol{p}, \boldsymbol{q}) \leq d(\boldsymbol{p}, \boldsymbol{t}) + \delta(\boldsymbol{t}, \boldsymbol{q}) \tag{A.1}$$

Proof. We proof for each of the 2 cases of Equation 4.1:

For case 1, where $\delta(\mathbf{p}, \mathbf{q}) = d(\mathbf{p}, \mathbf{q})$.

From 4.2:

$$d(\mathbf{t}, \mathbf{q}) \leq \delta(\mathbf{t}, \mathbf{q}) \tag{A.2}$$

$$d(\mathbf{p}, \mathbf{t}) + d(\mathbf{t}, \mathbf{q}) \leq d(\mathbf{p}, \mathbf{t}) + \delta(\mathbf{t}, \mathbf{q}) \tag{A.3}$$

From triangle inequality 4.5:

$$d(\mathbf{p}, \mathbf{q}) \leq d(\mathbf{p}, \mathbf{t}) + d(\mathbf{t}, \mathbf{q}) \tag{A.4}$$

Hence, from A.3 and A.4:

$$d(\mathbf{p}, \mathbf{q}) \leq d(\mathbf{p}, \mathbf{t}) + \delta(\mathbf{t}, \mathbf{q}) \tag{A.5}$$

$$\delta(\mathbf{p}, \mathbf{q}) \leq d(\mathbf{p}, \mathbf{t}) + \delta(\mathbf{t}, \mathbf{q}) \quad \square \tag{A.6}$$

For case 2, where $\delta(\mathbf{p}, \mathbf{q}) = C(\mathbf{q})$.

From 4.3:

$$C(\mathbf{q}) \leq \delta(\mathbf{t}, \mathbf{q}) \tag{A.7}$$

Recalling that distances are non-negative:

$$0 \leq d(\mathbf{p}, \mathbf{q}) \tag{A.8}$$

$$\delta(t, q) \leq d(\mathbf{p}, \mathbf{q}) + \delta(\mathbf{t}, \mathbf{q}) \tag{A.9}$$

Hence, from A.7 and A.9:

$$C(\mathbf{q}) \leq d(\mathbf{p},\mathbf{t}) + \delta(\mathbf{t},\mathbf{q}) \tag{A.10}$$
$$\delta(\mathbf{p},\mathbf{q}) \leq d(\mathbf{p},\mathbf{t}) + \delta(\mathbf{t},\mathbf{q}) \quad \square \tag{A.11}$$

□

Lemma 2. *Given $\boldsymbol{p}, \boldsymbol{q}, \boldsymbol{t} \in \mathcal{Z}$, where $C(\boldsymbol{p}) \leq C(\boldsymbol{q})$, the OPF-distance δ satisfies:*

$$\delta(x,p) \leq \delta(p,q) + \delta(x,q) \tag{A.12}$$

Proof. Again, we proof for each of the 2 cases of Equation 4.1:

For case 1, where $\delta(\mathbf{p},\mathbf{q}) = d(\mathbf{p},\mathbf{q})$.

From 4.2:

$$d(\mathbf{p},\mathbf{q}) \leq \delta(\mathbf{p},\mathbf{q}) \tag{A.13}$$
$$\text{and } d(\mathbf{t},\mathbf{q}) \leq \delta(\mathbf{t},\mathbf{q}) \tag{A.14}$$
$$d(\mathbf{p},\mathbf{q}) + d(\mathbf{t},\mathbf{q}) \leq \delta(\mathbf{p},\mathbf{q}) + \delta(\mathbf{t},\mathbf{q}) \tag{A.15}$$

From triangle inequality 4.5:

$$d(\mathbf{t},\mathbf{p}) \leq d(\mathbf{t},\mathbf{q}) + d(\mathbf{q},\mathbf{p}) \tag{A.16}$$

Alternatively, given that distance d is symmetric

$$d(\mathbf{t},\mathbf{p}) \leq d(\mathbf{p},\mathbf{q}) + d(\mathbf{t},\mathbf{q}) \tag{A.17}$$

Hence, from A.15 and A.17:

$$d(\mathbf{t},\mathbf{p}) \leq \delta(\mathbf{p},\mathbf{q}) + \delta(\mathbf{t},\mathbf{q}) \tag{A.18}$$
$$\delta(\mathbf{t},\mathbf{p}) \leq d(\mathbf{p},\mathbf{t}) + \delta(\mathbf{t},\mathbf{q}) \quad \square \tag{A.19}$$

For case 2, where $\delta(\mathbf{t},\mathbf{p}) = C(\mathbf{p})$.

From 4.3:

$$C(\mathbf{q}) \leq \delta(\mathbf{p},\mathbf{q}) \tag{A.20}$$
$$\text{and } C(\mathbf{q}) \leq \delta(\mathbf{t},\mathbf{q}) \tag{A.21}$$
$$C(q) \leq 2 \cdot C(q) \leq \delta(\mathbf{p},\mathbf{q}) + \delta(\mathbf{t},\mathbf{q}) \tag{A.22}$$

By hypothesis:

$$C(\mathbf{p}) \leq C(\mathbf{q}) \tag{A.23}$$

Hence, from A.22 and A.23:

$$C(\mathbf{p}) \leq \delta(\mathbf{p},\mathbf{q}) + \delta(\mathbf{t},\mathbf{q}) \tag{A.24}$$
$$\delta(\mathbf{t},\mathbf{p}) \leq \delta(\mathbf{p},\mathbf{q}) + \delta(\mathbf{t},\mathbf{q}) \quad \square \tag{A.25}$$

□

A.2 The OPF Tree

Lemma 3. *Given $\boldsymbol{p}, \boldsymbol{q}, \boldsymbol{t} \in \mathcal{Z}$, where $d(\boldsymbol{p}, \boldsymbol{t}) + r \leq M$ and $\delta(\boldsymbol{p}, \boldsymbol{q}) > M$, then $\delta(\boldsymbol{t}, \boldsymbol{q}) > r$.*

Proof.

From OPF Triangle Inequality (Lema 1):

$$\delta(\mathbf{p}, \mathbf{q}) \leq d(\mathbf{p}, \mathbf{t}) + \delta(\mathbf{t}, \mathbf{q}) \tag{A.26}$$
$$\delta(\mathbf{p}, \mathbf{q}) - d(\mathbf{p}, \mathbf{t}) \leq \delta(\mathbf{t}, \mathbf{q}) \tag{A.27}$$

From hypotheses:

$$d(\mathbf{p}, \mathbf{t}) + r \leq M \tag{A.28}$$
$$r \leq M - d(\mathbf{p}, \mathbf{t}) \tag{A.29}$$
$$\text{and } \delta(\mathbf{p}, \mathbf{q}) > M \tag{A.30}$$
$$0 < \delta(\mathbf{p}, \mathbf{q}) - M \tag{A.31}$$

Adding A.29 and A.31:

$$r < \delta(\mathbf{p}, \mathbf{q}) - d(\mathbf{p}, \mathbf{t}) \tag{A.32}$$

Thus, from A.27 and A.32:

$$r < \delta(\mathbf{t}, \mathbf{q}) \tag{A.33}$$

□

Lemma 4. *Given $\boldsymbol{p}, \boldsymbol{q}, \boldsymbol{t} \in \mathcal{Z}$, where $C(\boldsymbol{p}) \leq C(\boldsymbol{q})$, $\delta(\boldsymbol{p}, \boldsymbol{q}) \leq M$ and $\delta(\boldsymbol{t}, \boldsymbol{p}) - r > M$, then $\delta(\boldsymbol{t}, \boldsymbol{q}) > r$.*

Proof.

From OPF Triangle Inequality (Lema 2):

$$\delta(\mathbf{t}, \mathbf{p}) \leq \delta(\mathbf{p}, \mathbf{q}) + \delta(\mathbf{t}, \mathbf{q}) \tag{A.34}$$
$$\delta(\mathbf{t}, \mathbf{p}) - \delta(\mathbf{p}, \mathbf{q}) \leq \delta(\mathbf{t}, \mathbf{q}) \tag{A.35}$$

From hypotheses:

$$\delta(\mathbf{p}, \mathbf{q}) \leq M \tag{A.36}$$
$$-\delta(\mathbf{p}, \mathbf{q}) \geq -M \tag{A.37}$$
$$\text{and } \delta(\mathbf{t}, \mathbf{p}) - r > M \tag{A.38}$$
$$\delta(\mathbf{t}, \mathbf{p}) > M + r \tag{A.39}$$

Adding A.37 and A.39:

$$\delta(\mathbf{t}, \mathbf{p}) - \delta(\mathbf{p}, \mathbf{q}) > r \tag{A.40}$$

Thus, from A.35 and A.40:

$$r < \delta(\mathbf{t}, \mathbf{q}) \tag{A.41}$$

□

Bibliography

Autonuma: the other approach to numa scheduling. (2012, March). Retrieved from `http://lwn.net/Articles/488709/`

Chávez, E., Navarro, G., Baeza-Yates, R., & Marroquín, J. L. (2001). Searching in metric spaces. *ACM Computing Surveys (CSUR)*, *33*(3), 273–321.

Culquicondor, A., Castelo-Fernandez, C., & Papa, J. P. (2016). A New Parallel Training Algorithm for Optimum-Path Forest-based Learning. In *Progress in Pattern Recognition, Image Analysis, Computer Vision, and Applications* (pp. 192–199). Springer.

Dagum, L., & Enon, R. (1998). OpenMP: an industry standard API for shared-memory programming. *Computational Science & Engineering, IEEE*, *5*(1), 46–55.

Diniz, W. F., Fremont, V., Fantoni, I., & Nóbrega, E. G. (2017). An FPGA-based architecture for embedded systems performance acceleration applied to Optimum-Path Forest classifier. *Microprocessors and Microsystems*.

Duda, R. O., Hart, P. E., & Stork, D. G. (2012). *Pattern classification.* John Wiley & Sons.

Falcão, A. X., Stolfi, J., & de Alencar Lotufo, R. (2004). The image foresting transform: Theory, algorithms, and applications. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, *26*(1), 19–29.

Gaede, V., & Günther, O. (1998). Multidimensional access methods. *ACM Computing Surveys (CSUR)*, *30*(2), 170–231.

Hoberock, J., & Bell, N. (2010). *Thrust: A parallel template library.* version.

Indyk, P., & Motwani, R. (1998). Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proceedings of the thirtieth annual ACM symposium on Theory of computing* (pp. 604–613).

Iwashita, A. S., Papa, J. P., Falcao, A. X., Lotufo, R. A., de Araujo Oliveira, V. M., De Albuquerque, V. H. C., & Tavares, J. M. R. (2012). Speeding up optimum-path forest training by path-cost propagation. In *Pattern Recognition (ICPR), 2012 21st International Conference on* (pp. 1233–1236).

Iwashita, A. S., Papa, J. P., Souza, A., Falcão, A. X., Lotufo, R., Oliveira, V., ... Tavares, J. M. R. (2014). A path-and label-cost propagation approach to speedup the training of the optimum-path forest classifier. *Pattern Recognition Letters*, *40*, 121–127.

Iwashita, A. S., Romero, M. V., Baldassin, A., Costa, K. A., & Papa, J. P. (2014). Training optimum-path forest on graphics processing units. In *Computer Vision Theory and Applications (VISAPP), 2014 International Conference on* (Vol. 2, pp. 581–588).

Lichman, M. (2013). *UCI machine learning repository.* Retrieved from `http://archive.ics.uci.edu/ml`

Pacheco, P. (2011). *An introduction to parallel programming*. Elsevier.

Papa, J., Falcão, A., & Suzuki, C. (2014). LibOPF: A library for the design of optimum-path forest classifiers [Computer software manual]. (Software version 2.1 available at `http://www.ic.unicamp.br/~afalcao/LibOPF`)

Papa, J. P., Cappabianco, F. A., & Falcão, A. X. (2010). Optimizing optimum-path forest classification for huge datasets. In *Pattern Recognition (ICPR), 2010 20th International Conference on* (pp. 4162–4165).

Papa, J. P., & Falcão, A. X. (2008). A new variant of the optimum-path forest classifier. In *Advances in Visual Computing* (Vol. 5358, pp. 935–944). Springer Berlin Heidelberg.

Papa, J. P., Falcão, A. X., De Albuquerque, V. H. C., & Tavares, J. M. R. (2012). Efficient supervised optimum-path forest classification for large datasets. *Pattern Recognition*, *45*(1), 512–520.

Papa, J. P., Falcão, A. X., & Suzuki, C. T. (2009). Supervised pattern classification based on optimum-path forest. *International Journal of Imaging Systems and Technology*, *19*(2), 120–131.

Romero, M. V., Iwashita, A. S., Papa, L. P., Souza, A. N., & Papa, J. P. (2014). Fast optimum-path forest classification on graphics processors. In *Computer Vision Theory and Applications (VISAPP), 2014 International Conference on* (Vol. 2, pp. 627–631).

Uhlmann, J. K. (1991). Satisfying general proximity/similarity queries with metric trees. *Information processing letters*, *40*(4), 175–179.